INVESTIGATIONS IN TIME-DEPENDENT COMBINATORIAL OPTIMIZATION PROBLEMS AND THEIR APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

By

William B. Lassiter

In Partial Fulfillment of the
Requirements for the Degree
Doctor of Philosophy in Operations Research
within the H. Milton Stewart School
of Industrial and Systems Engineering

Georgia Institute of Technology

August 2020

INVESTIGATIONS IN TIME-DEPENDENT COMBINATORIAL OPTIMIZATION PROBLEMS AND THEIR APPLICATIONS

Approved by:

Dr. Martin Savelsbergh, Advisor
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Natashia Boland
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Alejandro Toriello
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Chelsea White
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Karen Feigh
Daniel Guggenheim School of Aerospace Engineering
Georgia Institute of Technology

Date Approved: May 15, 2020

There is nothing of which we are apt to be so lavish as of Time, and about which we ought to be more solicitous; since without it we can do nothing in this World.

William Penn

To William Carroll “Granddaddy” Lassiter and Connie Mack “Baba” Bowers,

two shining examples of how to live.

ACKNOWLEDGEMENTS

I would first and foremost like to thank my research advisor, Dr. Martin Savelsbergh, for being patient and flexible with me while continuing to hold me to a high standard. Composing this thesis was a long and sometimes winding journey, and I was not always the best or easiest advisee to work with. I would additionally like to thank Drs. Natashia Boland, Karen Feigh, Martijn IJtsma, and Amy Pritchett for their mentorship and collaboration on various pieces of this work. I also could not have made this five-year journey without the homework help, research input, and laughs provided by friends and colleagues in the department—Ahmad Baubaid, Reid Bishop, and Amanda Chu, to name a few.

Outside of the academic sphere, I would like to thank my parents, John and Julie, for their advice and unwavering support; my wonderful wife, Amy, for being there for me and putting up with all my nonsense; my uncle, Mack Bowers, for taking time out of his busy schedule to have lunch and listen to my ramblings on a regular basis; and my cousin, Benjamin Bowers, for splitting a cheap apartment with me for four years when he could easily have lived somewhere much nicer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

We explore three distinct but related combinatorial optimization problems involving time. Chapter 2 focuses on a time-indexed integer programming (IP) formulation for solving the Traveling Salesman Problem with Time Windows (TSPTW) exactly. The linear programming relaxation of this formulation provides a strong lower bound on its optimal value, making it an excellent candidate for use in branch-and-bound type solution algorithms, but the number of variables required to model large instances makes solving it very difficult and time-consuming. With this challenge in mind, we propose a Lagrangian duality-based variable elimination scheme designed to identify infeasible or provably sub-optimal time points that need not be included in the time-indexed IP model. Results for several instances from the literature are presented.

Chapter 3 shifts to a more applied setting, examining a scheduling problem currently faced by mission planners at NASA. One of the many problems that deep space travel (to Mars and beyond) presents is a significant lag time in communications between mission control and spacecraft as they move further away from Earth. At present, planners at mission control build minute-by-minute daily schedules for astronaut crews and update them in real time when circumstances require it, but with significant communications delays in deep space, astronauts will need increased autonomy, as well as automated assistance, in building and adjusting their own schedules. We present and discuss a prototype semi-autonomous scheduling system built in collaboration with colleagues from aerospace engineering to assist mission crews with rapid on-board re-planning in off-nominal (i.e. unanticipated) or emergency scenarios. We demonstrate the system's re-planning capabilities with two distinct case studies.

In Chapter 4, we examine a more general scheduling problem that is in some sense a natural offshoot of the work in Chapter 3. The problem of interest is that of scheduling jobs with start time-dependent deteriorating processing times on a single machine to minimize makespan (i.e. time to completion) when one or more fixed-length maintenance periods may be scheduled to mitigate

deterioration. In particular, the processing time for a job j with start time t has the linear form $p_j(t) = p_j + a_j t$, where p_j is a base processing time and $a_j > 0$ is a deterioration rate. We begin with a discussion of the problem's structure, follow this with two proposed IP formulations (one exact and one approximate), and finish with a computational analysis of several greedy heuristics designed to quickly obtain high-quality solutions.

CHAPTER 1
INTRODUCTION

Time is arguably the most valuable resource that we as humans possess. It is also unique among resources in that it cannot, at least in the literal sense, be saved, borrowed, or bought; only spent. It should therefore come as no surprise that many real-world decision problems—workforce and production scheduling, vehicle routing and transportation problems, and queuing systems, to name a few—compel us to optimize (or at the very least monitor) the expenditure of time. Each ensuing chapter of this thesis focuses on a specific combinatorial optimization problem involving time: we begin with the well-known Traveling Salesman Problem with Time Windows (TSPTW) in Chapter 2, follow this with an autonomous scheduling problem currently facing mission planners at NASA in Chapter 3, and finish with a single machine scheduling problem with time-dependent job processing times in Chapter 4.

TSPTW is a variant of the well-studied Traveling Salesman Problem (TSP) in which the salesman seeks a minimum-cost itinerary that, in addition to passing through each city i on his list exactly once, leaves i within a specified window $[e_i, \ell_i]$ of possible departure times. This adds an element of scheduling to the problem; we must now be cognizant not only of travel costs between cities but also of travel and departure times. Like TSP, TSPTW is NP-Hard, meaning it has no known polynomial-time solution algorithm, but it can be solved exactly using, among other techniques, integer programming (IP). In Chapter 2, we focus on a time-indexed IP formulation of TSPTW that uses binary variables y_{ij}^t to model the decision to depart a city i for another city j at time t. Discretizing and embedding time points within our arc-usage variables allows us to avoid the use of "big-M" constraints which are needed when modeling departure times as separate variables, provided that travel times between cities are integer-valued. The linear programming (LP) relaxation of this formulation generally provides a strong (i.e. tight) lower bound on the problem's

optimal value, making it an ideal candidate for use in a branch-and-bound or branch-and-cut solution algorithm. However, the strength of its LP relaxation is counterbalanced somewhat by the potentially massive number of variables y_{ij}^t that could arise when modeling problem instances involving a large number of cities and/or relatively wide time windows. For large enough instances, even the initial step of enumerating all time points (i, j, t) and loading their associated variables and constraints into a commercial solver like Gurobi becomes tedious and computationally expensive, and solving with an enumerative algorithm (if even possible) can take hours or days.

With this challenge in mind, we design and implement a Lagrangian duality-based variable elimination scheme designed to identify infeasible or provably sub-optimal time points that need not be included in the time-indexed IP model. Importantly, this scheme does not require enumerating all time points, instead uses (1) logic based on time window starts and ends and (2) knowledge of the special structure of a certain Lagrangian dual problem for TSPTW to identify "chunks" of time points that can be ignored when building the time-indexed formulation with no effect on its optimal solution/value. Computational results gathered over a number of instances from the literature are encouraging; in some cases more than 90% of time points are eliminated, in turn saving tremendous amounts of time in building and solving the IP.

In Chapter 3, we turn our attention to another time-constrained scheduling problem, this time in a much more applied setting courtesy of the National Aeronautics and Space Administration (NASA). Given the high levels of complexity and danger inherent in human spaceflight missions, astronauts' days in space are typically planned for them weeks or even months in advance, and down to the very minute, by teams of expert planners. Today, when something goes wrong or a change of plan is required in a setting like the International Space Station, experts on the ground can quickly initiate a re-planning operation and communicate the new plan to astronauts to get them back on track. However, as NASA gears up for a proposed manned mission to Mars, its mission planners are struggling with how best to handle short-term re-planning in the face of distance-induced communications delays that, depending on the relative positioning of Earth and Mars,

may run up to 24 minutes one way. As most astronauts are not well-versed in mission planning or resource management, the general consensus at NASA is that crews on missions to Mars and deeper space will need some form of on-board planning assistance to provide them with a certain amount of autonomy in managing and adjusting plans.

To this end, we teamed with colleagues in aerospace engineering to create a prototype scheduling system to assist mission crews with rapid on-board re-planning in off-nominal (i.e. unanticipated) or emergency scenarios. The system consists of a plan optimization tool (our main contribution) and a work simulator (built by our colleagues), both coded in C++, that work together over a unified plan modeling framework to aid astronauts in selecting new, feasible plans in a timely manner when the current plan is no longer acceptable.

Our optimization tool, which works with a simplified version of a plan in the mold of a machine scheduling problem, employs a jump-based local search algorithm to improve the plan with respect to one or more specified objectives and/or re-establish its feasibility in off-nominal circumstances. This plan simplification is necessary to avoid detailed, computationally expensive feasibility checks and objective evaluations at each step of the search, but can also lead to unforeseen issues with a proposed new plan. Thus prospective plans are passed to the work simulator for detailed evaluation, the results of which are taken as feedback by the optimization tool, potentially triggering adjustments to the simplified plan's parameters and more searching until a "good" solution is found. Chapter 3 begins with a description of our plan modeling framework before diving in to the ins and outs of our scheduling system and finally presenting some results obtained by running a day's worth of activities from one of NASA's NEEMO missions through our system.

In Chapter 4, we examine a more general scheduling problem that is in some sense a natural offshoot of the work in Chapter 3. One of the scheduling objectives we chose to focus on in the context of human spaceflight was astronaut fatigue (and by extension workload balance among astronauts). In modeling fatigue, we assume that astronauts having just completed several intensive consecutive tasks (e.g. as part of a spacewalk) will take longer to complete a subsequent task

compared to when they had just rolled (or floated) out of bed. This fatigue effect can be more generally modeled as a single machine scheduling problem in which we simulate a machine's deteriorating performance over time using time-dependent job processing times. Our particular problem of interest defines the processing time for a job j as

$$p_j(t) := p_j + a_j t \ ,$$

where t is the time at which the job begins processing relative to the start of the schedule, p_j is how long the job takes to process when the machine is fresh (i.e. at time 0), and $a_j > 0$ is the job's deterioration rate. We elect to use job-specific deterioration rates to allow for variation in perceived job difficulty; for example, a chat with schoolchildren via satellite on the ISS should be much less physically and emotionally taxing than a series of critical station repairs.

If our goal is to minimize schedule makespan (i.e. time to completion), the problem as stated is an easy one: simply scheduling jobs in ascending order of the ratio $\frac{p_j}{a_j}$ is optimal. We consider a more interesting, albeit more difficult, makespan minimization scenario in which we are allowed to schedule one or more maintenance periods between jobs that effectively return the machine to its initial operating state (i.e. reset the clock to 0). While much work has been done on single machine scheduling problems in this vein, this particular problem variant (with job-specific processing times *and* deterioration rates) has, to our knowledge, never been addressed in the literature. Chapter 4 begins with a discussion of solution methods for related problem variants, proceeds to formulate the problem as an integer program and offer some thoughts on its hardness, and finishes with an examination of several heuristic methods for arriving at good solutions.

While the problems explored in the chapters that follow are, despite all involving time in one way or another, largely distinct from one another, all of the research presented herein is born out of a desire to expand the realm and reach of operations research (OR). In Chapter 2, this means introducing new methods for pre-processing a powerful IP formulation whose weakness lies in the

potentially massive number of time points (and corresponding variables and constraints) needed to assemble it. Pre-processing routines for TSPTW based on node precedence relationships and time windows have been explored in prior work, but our use of arc and arc time window-based pre-processing techniques is, to our knowledge, novel. Furthermore, while a more traditional reduced-cost fixing method would require enumerating and checking all time point variables, our proposed method uses the results of a simple dynamic programming algorithm to identify whole intervals of unnecessary time points without enumeration.

In Chapter 3, we look to expand the reach of OR by modeling a current quandary in the realm of human spaceflight as a special class of machine scheduling problems. Up to now, planning for human spaceflight has focused almost exclusively on schedule feasibility; our prototype system explores the possibility of improving a feasible schedule with respect to one or more planning objectives via local search. Our plan modeling and adjustment frameworks also provide an excellent foundation for the development of more advanced and/or automated planning algorithms and workflows.

Finally, in Chapter 4 we expand the realm of OR with work on a little explored variant of the single machine scheduling problem with time-dependent processing times and maintenance. Allowing both base processing times p_j and deterioration rates a_j to vary by job enables more realistic modeling of scheduling scenarios that involve jobs of different lengths and difficulty levels. The techniques we have developed may also have more general uses in combinatorial optimization, e.g. in solving weighted set cover problems with a cardinality constraint.

CHAPTER 2

A LAGRANGIAN DUALITY-BASED APPROACH FOR PRE-PROCESSING THE TRAVELING SALESMAN PROBLEM WITH TIME WINDOWS

2.1 Introduction

The Traveling Salesman Problem (TSP) is the celebrated and well-studied problem of finding a minimum-cost route through a given set of cities that visits each of them exactly once and terminates where it began (such a route is referred to as a tour). The Traveling Salesman Problem with Time Windows (TSPTW) is a variant of the TSP that incorporates travel times between cities (which may or may not be symmetric) and requires that a tour must depart a city within a certain time window whose left and right endpoints are the earliest and latest times, respectively, at which the route can depart, and that the tour must be completed by some time T. If a tour arrives at a city prior to the start of its time window, it must wait until that time to depart; additionally, departing after the end of the window is not permitted. The aim is generally to either minimize the start-to-finish time (makespan) of a tour or to minimize the sum of travel times between cities.

Applications of the TSPTW are wide-ranging: it can be used to solve time-sensitive machine scheduling [1], vehicle routing in automated manufacturing systems [2, 3], postal and bank delivery, and shipping [2] problems, among others. One paper even applies an extension of TSPTW to solve a home meal delivery problem [4].

By virtue of being a generalization of the TSP, the TSPTW is $\mathcal{NP}$-Hard; Savelsbergh [5] proved that even the problem of finding a feasible solution is $\mathcal{NP}$-Complete. The earliest proposed computational methods for solving the problem were those of Christofides, Mingozzi and Toth [6], who presented a dynamic programming-based branch-and-bound framework, and Baker [7], who offered a linear programming-based branch-and-bound approach whose variables are city depar-

ture times. Both approaches were only aimed at minimizing the overall travel time (makespan) of a tour.

In the years following these first attempts, dynamic programming has remained one of the most popular approaches to solving the TSPTW. Dumas et al. [8] and Mingozzi, Bianco, and Ricciardelli [9] devised means for greatly reducing the state space of a dynamic programming formulation, with the former group using an elimination procedure based on time window constraints and the later a more general version of the technique used in [6]. Both of these techniques were now also able to minimize the sum of the travel times in a tour. Balas and Simonetti [10] explored a slightly different dynamic programming approach based on precedence relationships that can be used to solve the problem to optimality quickly under certain conditions.

Researchers have also experimented with a profusion of other approaches not based in dynamic programming. Langevin et al. proposed a two-commodity flow formulation and a branch-and-bound scheme to solve it that could handle both problem objectives mentioned earlier. The Miller-Tucker-Zemlin (MTZ) [11] formulation for the standard TSP can easily be adapted to model the TSPTW, but must use so-called "big-M" constraints that diminish the strength of its linear programming relaxation. Ascheuer et al. [12] introduced a polyhedral integer programming formulation for TSPTW that used infeasible path constraints in place of the MTZ formulation's big-M constraints and compared favorably to it. Pesant et al. [13] were among the first groups to offer a constraint programming approach for solving the problem. Focacci et al. [14] built on this by using classical operations research techniques such as Lagrangian relaxation, reduced cost-fixing, and cutting planes to reduce the size of the problem before solving it with constraint programming.

More recently, another class of integer programming models has been explored as a means of solving the problem–a class focused around time-expanded networks. Albiach et al. [15] offered a time-indexed formulation for the asymmetric TSPTW with time-dependent travel times that Gunluk et al.[16] adapted for the symmetric TSPTW with constant travel times. The formulation boasts a very strong linear programming (LP) relaxation but requires relatively large numbers of variables

and constraints. Neither group attempted to solve their fully time-expanded model directly, and to our knowledge no one has done so. The authors in [16] instead propose branch-and-cut methods for solving a smaller model in which time windows have been partitioned into "buckets" instead of individual time points. Boland et al. (cite Boland) solve a time-expanded network model similar to the one presented in [16] using partially time-expanded networks, again avoiding using the full time expansion.

The current computational gold standard for solving the TSPTW to optimality is a method proposed by Baldacci et al. [17] in 2012. The method builds on that of [6] by introducing a new, stronger tour-based relaxation of the TSPTW and solving its dual with column generation to obtain a strong lower bound on the problem's optimal value, and then using this lower bound in a dynamic programming algorithm to solve the problem exactly.

This chapter considers the fully time-expanded formulation of the TSPTW using two interwoven techniques for identifying and removing from consideration time points at which an optimal tour could not possibly visit a city. The first is an arc-based preprocessing routine designed to shrink and split up time windows for individual cities based on time windows of neighboring cities. The second is a Lagrangian duality-based reduced cost variable fixing method designed to eliminate additional time points that don't have the potential to improve a solution. The key contribution of this chapter lies in our technique's effectiveness in greatly reducing the size of the fully time-expanded model without ever explicitly enumerating all time points of the problem's full time expansion.

The chapter is organized as follows. In Section 2 we formally state the problem, present a fully time-expanded integer programming model that can be used to solve it, and provide a high-level look at our solution method. In Section 3 we discuss the preprocessing techniques utilized by this solution method. Section 4 details our use of a Lagrangian relaxation of the problem to eliminate variables using reduced cost variable fixing. Section 5 reports computational results for our technique on various instances from the literature. Finally, Section 6 offers conclusions and

some ideas for future work.

2.2 Problem Statement and Formulation

We now provide a formal statement of the asymmetric TSPTW (ATSPTW). We are given a set $\{0, 1, 2, ..., n-1\}$ of n nodes (or cities), with each node i assigned a time window $W_i = [e_i, \ell_i]$, where e_i and ℓ_i denote the earliest and latest times, respectively, at which a tour is allowed to depart that node. Node 0 is designated as the "depot" node with an initial given time window $W_0 = [0, H]$, where H is the end of the problem's time horizon. However, in light of the way a time window is defined above, we can drastically shrink the time window for 0 a priori by creating a copy of it, which we will call $0'$, to serve as a terminus. We will hereafter refer to the set $\{0, 1, 2, ..., n-1\} \cup \{0'\}$ as $\mathcal{N}$. The time window for 0 can be reduced to $W_0 = [0, \min_{i \in \mathcal{N} \setminus \{0\}}\{\ell_i - \tau_{0i}\}]$, and the time window for $0'$ is set as $W_{0'} = [\max_{i \in \mathcal{N} \setminus \{0'\}}\{e_i + \tau_{i0}\}, H]$.

Now, let $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ be a directed graph on these nodes with arc set $\mathcal{A}$, where each arc (i, j) connects two distinct nodes i and j at a given cost of $c_{ij} \geq 0$ and with a given travel time of $\tau_{ij} > 0$. Neither costs nor travel times are necessarily symmetric. We assume that all nodes i given in an instance have $e_i \geq e_0 + \tau_{0i}$ and $\ell_i \leq \ell_0 - \tau_{i0}$, that no arcs enter 0 or leave $0'$, and that all given time window and travel time data is integral, which allows W_i to be more simply expressed as $W_i = \{e_i, e_i + 1, \ldots, \ell_i\}$. In this context, a feasible tour is a permutation of $\mathcal{N}$ (always beginning with 0 and ending with $0'$) in which for all nodes $j \in \mathcal{N}$ the departure time from node j falls within its time window W_j, where departure time is taken to be the maximum of e_j and the departure time from the previous node in the sequence, say i, plus travel time τ_{ij}. In light of this, we can safely assume a priori that the arc set $\mathcal{A}$ includes only arcs (i, j) for which $e_i + \tau_{ij} \leq \ell_j$, as arcs that do not meet this requirement could not be traversed by a feasible tour. It should also be noted that for each arc $(i, j) \in \mathcal{A}$ we can deduce from node time windows for i and j a time window during which we can depart along (i, j). Such a window is initially given as

$$W_{(i,j)} = [e_{(i,j)}, \ell_{(i,j)}] := [e_i, \min\{\ell_i, \ell_j - \tau_{ij}\}].$$

To solve the ATSPTW is to find an optimal tour—a feasible tour as described above with minimum cost. As can be surmised from our notion of departure time, a tour is permitted to "wait", i.e. to arrive at a node i strictly prior to time e_i and then to depart precisely at e_i. Numerous integer programming models have been proposed for the ATSPTW; our focus will be on a time-indexed formulation discussed in [16] that is based in binary decision variables y_{ij}^t, where y_{ij}^t takes value 1 if the tour departs city i for city j at time $t \in W_i$ and is 0 otherwise. A particular y_{ij}^t variable is included in the model only if one can depart from i at time t and reach j by the close of its time window, i.e. only if $t + \tau_{ij} \leq \ell_j$.

Before explicitly stating the model we introduce some useful sets around which its constraints are designed. First, for a given node $i \in \mathcal{N}$, let $\delta^+(i)$ denote the set of nodes j for which there exists an arc from i to j, i.e. $\delta^+(i) = \{j \in \mathcal{N} : (i,j) \in \mathcal{A}\}$. Similarly, let $\delta^-(i) = \{j \in \mathcal{N} : (j,i) \in \mathcal{A}\}$. Second, for given nodes i, j and time $t \in W_i$, let $I_j(i,t)$ represent the set of times at which a tour could have departed j for i given that it departs from i at time t, i.e. $I_j(i,t) = \{s \in W_j : \max\{s + \tau_{ji}, e_i\} = t\}$. For $t > e_i$, this set will simply be the singleton $t - \tau_{ij}$ (assuming this time point lies in W_j) whereas for $t = e_i$ it will be the collection of times at which a tour could have left j to arrive at i at or prior to time e_i. Defining $I_j(i,t)$ in this way embeds an assumption of no unnecessary waiting at a node prior to departure, which may eliminate feasible and even optimal solutions; however, moving departure times up in any such solution to eliminate unnecessary waiting yields an equivalent solution with the same collective travel time between nodes. The model presented below is not identical to the one presented in [16], as it does not include the binary variables x_{ij} that the authors use to indicate arc usage, but is otherwise

equivalent:

$$\textbf{min} \quad \sum_{i \in \mathcal{N} \setminus \{0'\}} \sum_{t \in W_i} \sum_{j \in \delta^+(i)} c_{ij} y_{ij}^t$$

$$\textbf{s.t.} \quad \sum_{t \in W_i} \sum_{j \in \delta^+(i)} y_{ij}^t = 1 \quad \forall i \in \mathcal{N} \setminus \{0'\} \tag{2.1}$$

$$\sum_{j \in \delta^-(i)} \sum_{s \in I_j(i,t)} y_{ji}^s = \sum_{j \in \delta^+(i)} y_{ij}^t \quad \forall i \in \mathcal{N} \setminus \{0, 0'\},\ \forall t \in W_i \tag{2.2}$$

$$\sum_{t \in W_{0'}} \sum_{j \in \delta^-(0')} \sum_{s \in I_j(0',t)} y_{j0'}^s = 1 \tag{2.3}$$

$$y_{ij}^t \text{ binary } \forall i \in \mathcal{N} \setminus \{0'\},\ \forall t \in W_i,\ \forall j \in \delta^+(i) \tag{2.4}$$

Constraints (2.1) ensure that the tour departs each node exactly once, constraints (2.2) enforce flow balance, and constraint (2.3) forces the tour to end at $0'$.

The strength of this formulation is in the tightness of lower bound given by its LP relaxation; more compact models that do not index arc variables by time but instead maintain a separate variable for departure time are nonlinear, and their linearizations provide much weaker lower bounds. See [16] for a more detailed discussion and related proof. The model's major weakness, however, lies in the number of variables and constraints that it requires, which grows very large as average time window width increases.

As mentioned earlier in the chapter, our approach to combating this weakness is to reduce the size of the problem by eliminating time points that will not be used in every optimal tour from node time windows (and thus eliminating their respective variables and constraints from the model above). We accomplish this in two distinct ways. First, we build on a time window preprocessing routine given in [12] for identifying time points that cannot be used in any feasible tour by introducing arc-based conditional logic to shrink time windows even further. Second, we consider a Lagrangian relaxation of the model above to implicitly, and efficiently, solve its LP

relaxation, allowing us to employ a variation on the classical technique of reduced cost variable fixing to identify time points whose use will not improve a solution and may therefore also be removed from consideration.

Our overall solution method is predicated on moving back and forth between these two routines until enough time points have been eliminated to solve the given model relatively quickly. Each routine is discussed in more detail in the sections that follow.

2.3 Time Window Preprocessing

Prior to formally stating this first routine, we must introduce the notion of precedence relationships between nodes. We say that node i precedes node j (written $i \prec j$), or equivalently j succeeds i, if it can be logically deduced that i must be visited before j in any feasible tour. Thus we define sets $Pred(i) := \{j \in \mathcal{N} \setminus \{0, 0'\} : j \prec i\}$ and $Succ(i) := \{j \in \mathcal{N} \setminus \{0, 0'\} : i \prec j\}$ as the sets of nodes that must precede and succeed node i in the tour, respectively. Two simple checks that can be used to initially populate and to update these sets are:

1. If for nodes i and j we have that $e_j + \tau_{ji} > \ell_i$, then it must be that $i \prec j$.

2. Transitivity: if for nodes i, j, k we have that $i \prec j$ and $j \prec k$, then it must be that $i \prec k$.

As soon as it is determined that a node j must precede a node i, the arc (i, j) can be removed from $\mathcal{A}$ (and the sets $\delta^+(i)$ and $\delta^-(j)$ updated accordingly). Additionally, as soon as the set $Pred(i)$ is determined to be non-empty, the arc $(0, i)$ may be removed from $\mathcal{A}$. Similarly we may remove arc $(i, 0')$ as soon as $Succ(i)$ becomes non-empty. It should be noted that by the nature of our problem setup it will certainly be the case that 0 precedes all other cities and $0'$ succeeds all other cities, and there is no need to explicitly include them in $Pred$ and $Succ$ sets.

The preprocessing steps below (1) reduce the size of node time windows by identifying time points that cannot or need not be used in any solution, and (2) identify and eliminate arcs that cannot or need not be traversed in any solution. Doing both of these things has the potential to greatly reduce the difficulty of solving an instance of TSPTW while preserving at least one of its

optimal solutions. We have separated the routine into three distinct stages: Stage 1 considers for each node i the starts and ends of time windows for nodes in $\delta^+(i)$, $\delta^-(i)$, $Pred(i)$, and $Succ(i)$ and uses them to shrink its time window; Stage 2 uses logic based on node precedence relationships to eliminate arcs from $\mathcal{A}$; and Stage 3 again considers each node i and creates "holes" in time windows (time points $t \neq e_i$ for which there is no node j such that $t - \tau_{ji} \in W_j$).

2.3.1 Refine node time windows

Using Node-based Logic ([12],[16])

We iterate through the following 4 steps, performing each step for each node $i \in \mathcal{N} \setminus \{0, 0'\}$ before moving to the next step, until no changes are made in a full 4-step cycle. Steps 1 and 4 eliminate *infeasible* time points from the time window for node i, i.e. time points at which a feasible tour could not possibly arrive at i, while steps 2 and 3 eliminate *redundant* time points, i.e. time points whose removal will not affect the existence of an optimal tour.

1. A feasible tour must travel along some arc in $\delta^-(i)$ and thus cannot arrive at i prior to the earliest time i can be reached along one of these arcs. A feasible tour must also visit all nodes in $Pred(i)$ before visiting i. Therefore, update

$$e_i \leftarrow \max\left\{e_i, \min_{j \in \delta^-(i)}\{e_j + \tau_{ji}\}, \max_{j \in Pred(i)}\{e_j + \tau_{ji}\}\right\}$$

2. A feasible tour need not depart i at a time such that it would arrive prior to time e_j for all $j \in \delta^+(i)$; once arriving at any j the tour would be forced to wait until e_j to depart for another node, meaning i's time window can be reduced to eliminate this guaranteed wait time without affecting the existence of an optimal tour. Therefore, update

$$e_i \leftarrow \max\left\{e_i, \min\left\{\ell_i, \min_{j \in \delta^+(i)}\{e_j - \tau_{ij}\}\right\}\right\}.$$

3. A feasible tour need not depart i at any time after the latest time it can arrive at i from a node $j \in \delta^-(i)$; arriving at node i within its time window and waiting any amount of time before departing unnecessarily increases the completion time of the tour. Therefore, update

$$\ell_i \leftarrow \min\left\{\ell_i, \max\left\{e_i, \max_{j\in\delta^-(i)}\{\ell_j + \tau_{ji}\}\right\}\right\}.$$

4. A feasible tour must travel along some arc in $\delta^+(i)$ and thus must depart i at or prior to the latest time one of these arcs can be traversed. A feasible tour must also visit all nodes in $Succ(i)$ after visiting i. Therefore, update

$$\ell_i \leftarrow \min\left\{\ell_i, \max_{j\in\delta^+(i)}\{\ell_j - \tau_{ij}\}, \min_{j\in Succ(i)}\{\ell_j - \tau_{ij}\}\right\}$$

Using Arc-based Logic

Recall that for each arc $(i, j) \in \mathcal{A}$ we can deduce a time window

$$W_{(i,j)} = [e_i, \min\{\ell_i, \ell_j - \tau_{ij}\}].$$

The left and right endpoints of $W_{(i,j)}$ may be updated as follows:

1. If node j has predecessors other than i, each of them must have been visited prior to node i if arc (i, j) is to be used. Therefore if the set $Pred(j) \setminus \{i\}$ is non-empty, update

$$e_{(i,j)} \leftarrow \max\left\{e_{(i,j)}, \max_{k\in Pred(j)\setminus\{i\}}\{e_k + \tau_{ki}\}\right\}.$$

2. If node i has successors other than j, the tour must be able to reach each of them after it

visits j if arc (i,j) is to be used. Therefore if the set $Succ(i) \setminus \{j\}$ is non-empty, update

$$\ell_{(i,j)} \leftarrow \min\left\{\ell_{(i,j)}, \min_{k \in Succ(i)\setminus\{j\}} \{\ell_k - \tau_{jk} - \tau_{ij}\}\right\}.$$

If after these two steps any arc (i,j) has $e_{(i,j)} > \ell_{(i,j)}$, it may be deleted from $\mathcal{A}$. Node time windows can now be updated as

$$W_i \leftarrow \bigcup_{j \in \delta^+(i)} W_{(i,j)}.$$

This arc window refinement may expose holes in time windows during which a tour cannot feasibly depart i (for instance, if $e_{(i,j)} = e_i$ for some j and $e_{(i,k)} \geq e_i + 2$ for all other nodes k), so that they can no longer be represented as a single closed interval. When this is the case, we represent W_i as a collection of m_i disjoint closed intervals (where there are $m_i - 1$ holes), whose left and right endpoints we will denote by $e_i^1 = e_i, e_i^2, \ldots, e_i^{m_i}$ and $\ell_i^1, \ell_i^2, \ldots, \ell_i^{m_i} = \ell_i$ so that we have

$$W_i = \bigcup_{k=1}^{m_i} \{e_i^k, e_i^k + 1, \ldots, \ell_i^k\}.$$

2.3.2 Delete Arcs ([12],[16])

Using the two simple checks discussed at the beginning of this section, we update precedence relationships based on refined time windows and then carry out the three steps below:

1. Delete all arcs (i,j) for which $j \prec i$.
2. Delete all arcs (i,j) for which there is a k such that $i \prec k \prec j$.
3. Delete all arcs (i,j) for which there is a k such that

$$e_k + \tau_{ki} + \tau_{ij} > \ell_j \text{ and } e_i + \tau_{ij} + \tau_{jk} > \ell_k.$$

If additionally we find that $e_i + \tau_{ik} + \tau_{kj} > \ell_j$, we have that k cannot be visited before the

arc (i, j) is used, k cannot be visited between visits to i and then j, and k cannot be visited after the arc (i, j) is used, thus allowing us to deduce that j must precede i.

Stages 1 and 2 are repeated until no changes are made in a full 2-stage cycle.

2.3.3 Identify "Holes" ([16])

Once stages 1 and 2 are complete, we identify time points for each node i when a feasible tour could not possibly visit that node. This is done by determining for each $j \in \delta^-(i)$ a window of times at which i can be reached using arc (j, i) and overlaying this window on W_i; once this has been done for every j, we can remove any time points in W_i not covered by the overlay.

The resulting time windows are then inserted into our second, Lagrangian duality-based time point elimination technique.

2.4 Lagrangian Relaxation and Reduced Cost Variable Fixing

The second routine we employ to reduce the problem's size takes advantage of the special structure of the Lagrangian relaxation $z_{LR}(\mathbf{u})$ obtained by moving constraints (2.1) (save the one for node 0) into the objective function with a penalty vector $\mathbf{u}$ of Lagrange multipliers, as given below:

$$z_{LR}(\mathbf{u}) = \sum_{i \in \mathcal{N} \setminus \{0,0'\}} u_i \; + z_{SP}(\mathbf{u})$$

where

$$
\begin{aligned}
z_{SP}(\mathbf{u}) = \quad & \mathbf{min}_y \quad \sum_{i \in \mathcal{N} \setminus \{0'\}} \sum_{t \in W_i} \sum_{j \in \delta^+(i)} (c_{ij} - u_i) y_{ij}^t \\
& \mathbf{s.t.} \quad \sum_{t \in W_0} \sum_{j \in \delta^+(0)} y_{0j}^t = 1 \\
& \qquad \sum_{j \in \delta^-(i)} \sum_{s \in I_j(i,t)} y_{ji}^s = \sum_{j \in \delta^+(i)} y_{ij}^t \quad \forall i \in \mathcal{N} \setminus \{0, 0'\},\ \forall t \in W_i \\
& \qquad \sum_{t \in W_{0'}} \sum_{j \in \delta^-(0')} \sum_{s \in I_j(0',t)} y_{j0'}^s = 1 \\
& \qquad y_{ij}^t \text{ binary } \forall i \in \mathcal{N} \setminus \{0'\},\ \forall t \in W_i,\ \forall j \in \delta^+(i)
\end{aligned}
$$

As the constraints stipulating that each node be visited exactly once have been removed, the minimization portion of $z_{LR}(\mathbf{u})$, which for brevity we represent as $z_{SP}(\mathbf{u})$, is now simply a Shortest Path Problem with Time Windows (SPPTW), which can be solved without enumerating all time points using dynamic programming. Additionally, as we will prove in Section 4.2, the optimal objective value of the Lagrangian dual (LD) problem given by

$$z_{LD} = \max_{\mathbf{u}} \ z_{LR}(\mathbf{u}) \tag{2.5}$$

equals that of the LP relaxation of the formulation presented in Section 2, which allows us to use feasible solutions of the Lagrangian dual problem in a reduced cost variable fixing setting.

2.4.1 Solving the Lagrangian Dual Problem

A simple, straightforward approach to solving the Lagrangian dual problem that has performed remarkably well in our computational experience is the use of a subgradient algorithm. The algorithm we have chosen to employ to solve for z_{LD} is largely based on one discussed in Wolsey [18]; if we let D represent the constraint matrix for constraints (2.1), it can be expressed as follows:

Algorithm 1 Subgradient Algorithm for the Lagrangian Dual

Initialization: vector $\mathbf{u}^0$ of dual multipliers, predetermined maximum number of iterations K, target value z, threshold value ϵ

for $k = 0, \ldots, K-1$ **do**

Solve $z_{LR}(\mathbf{u}^k)$ to obtain optimal $\mathbf{y}(\mathbf{u}^k)$.

if $|z - z_{LR}(\mathbf{u}^k)| < \epsilon$ **then**

BREAK, return $\mathbf{u}^k$.

else

Set $\mathbf{u}^{k+1} = \mathbf{u}^k + \mu_k \left(\mathbf{1} - D\mathbf{y}(\mathbf{u}^k)\right)$.

$k \leftarrow k + 1$.

The statement of the algorithm is fairly straightforward; however, a proper implementation requires that we select an efficient method for solving $z_{LR}(\mathbf{u}^k)$ and an appropriate step size μ_k at the k^{th} iteration.

Solving the Lagrangian Relaxation Problem

As all but one of constraints (2.1) have been moved into its objective function, $z_{SP}(\mathbf{u})$ now consists only of a "flow out equals 1" constraint for node 0, flow-balance constraints for all nodes i in $\mathcal{N} \setminus \{0, 0'\}$, and a "flow in equals 1" constraint for node $0'$, meaning we are left with what is essentially a SPPTW with source and sink nodes 0 and $0'$, respectively. The SPPTW is a special case of the Shortest Path Problem with Resource Constraints (SPPRC) in which cost and time are the only monitored resources, and time is the lone constrained resource. The time needed to travel between nodes i and j is still τ_{ij}, mirroring those of the TSPTW; the cost, however, is now $c_{ij} - u_i$. It is important to note here that while arc costs defined in this way may become negative and give rise to negative cycles, such cycles may not be traversed indefinitely due to the presence of time windows and the monotonic increase of the time resource.

Numerous methods have been developed for solving the SPPTW efficiently. Dynamic programming, which has been used extensively to this end, is our chosen approach; we employ a path-based labeling algorithm whose generic framework is given in [19]. The algorithm works by starting from the source node and extending paths in all feasible directions, assigning a label to each path to tie it to other paths with the same prefix (sequence of nodes in the path up to but not including the path's end node) and to keep track of accrued cost and travel time. Paths are compared against one another using their labels, and paths that do not have the potential to yield a Pareto-optimal solution are discarded according to dominance rules. At its end, the algorithm outputs a subset of solutions of the SPPTW from which a minimum-cost solution path can easily be gleaned.

Before formally stating the algorithm, we must introduce some additional notation. Given a path $P = (v_0, v_1, ..., v_p)$ of length p, let $P_i = (v_0, v_1, ..., v_i)$ denote the truncated path of length i and let $v(P)$ denote the last node visited by P. For each path, we maintain a 2-dimensional resource vector $T(P)$ that keeps track of both the time and cost accrued along the path P. The time component is calculated recursively as

$$
\begin{aligned}
T^1(P) &= max\{e_{v_p}, T^1(P_{p-1}) + \tau_{v_{p-1},v_p}\},\\
T^1(P_{p-1}) &= max\{e_{v_{p-1}}, T^1(P_{p-2}) + \tau_{v_{p-2},v_{p-1}}\},\\
&\vdots\\
T^1(P_0) &= 0
\end{aligned}
$$

and the cost component is calculated as

$$T^2(P) = \sum_{i=0}^{p-1}(c_{v_i,v_{i+1}} - u_{v_i}).$$

The labeling algorithm populates and modifies two sets of paths, a set $\mathcal{U}$ of useful paths and a set

$\mathcal{P}$ of processed paths. Its formal statement is given below:

Algorithm 2 Labeling Algorithm for the SPPTW

Initialization: Set $\mathcal{U} = \{(0)\}, \mathcal{P} = \emptyset$.
while $\mathcal{U} \neq \emptyset$ **do**
  Choose path $Q \in \mathcal{U}$ and remove Q from $\mathcal{U}$.
  for all arcs $(v(Q), w) \in \mathcal{A}$ **do**
    if $T^1(Q) + \tau_{v(Q),w} \leq \ell_w$ and $w \neq 0'$ **then**
      Add (Q, w) to $\mathcal{U}$
    Add Q to $\mathcal{P}$
  for each $v \in \mathcal{N} \setminus \{0\}$ **do**
    Compare all paths $P \in \mathcal{U} \cup \mathcal{P}$ with $v(P) = v$, and discard all paths Q for which there is a path P with $T(P) \leq T(Q)$ (i.e. $T^1(P) \leq T^1(Q)$ and $T^2(P) \leq T^2(Q)$).
Identify paths $P \in \mathcal{P}$ with $v(P) = 0'$ and choose the one with minimum cost.

The total cost of the solution path plus the sum of the components of $\mathbf{u}$ yields the value of $z_{LR}(\mathbf{u})$, which we can then use to define our step size μ_k .

Step Size

We are now able to incorporate the total cost of traversing the given solution path, along with how many times it visited each node i, into the calculation of μ_k. Our choice for step size, which mirrors an option presented in [18], requires a lower bound $\underline{z}$ on z_{LD} and defines μ_k as

$$\mu_k = \frac{\underline{z} - z_{LR}(\mathbf{u}^k)}{\|\mathbf{1} - D\mathbf{y}(\mathbf{u}^k)\|_2^2}.$$

Defining μ_k in this way guarantees (see [18]) that $\{z_{LR}(\mathbf{u}^k)\}_{k=0}^{\infty}$ converges to z_{LD}. The solution path obtained from our labeling algorithm provides all the necessary information here (save $\underline{z}$), as $z_{LR}(\mathbf{u}^k)$ is computed as described at the end of the previous subsection, and the i^{th} component of the vector $\mathbf{1} - D\mathbf{y}(\mathbf{u}^k)$ is equivalent to 1 minus the number of times that it visits node i.

Once we have obtained a vector $\overline{\mathbf{u}}$ from our subgradient algorithm, we are able to implement a variant of reduced cost variable fixing that is effective in eliminating additional variables in our

original IP formulation.

2.4.2 Reduced Cost Variable Fixing Using the Lagrangian Dual

As was mentioned in Section 2, the strength of the time-indexed formulation we are working with lies primarily in the tightness of the lower bound obtained from solving its LP relaxation. The closer a lower bound provided by solving a relaxation of an integer program (IP) is to the IP's optimal value, the more effective reduced cost variable fixing can be. Reduced cost variable fixing in its most basic form uses a problem's optimal LP value and reduced costs of nonbasic variables together with an upper bound on its optimal IP value to determine if any of these nonbasic variables can be permanently fixed to zero. Specifically, given optimal LP value z_{LP}, a nonbasic variable x_j and its reduced cost $\bar{c}_j$, and an upper bound U on the problem's optimal IP value, if $z_{LP} + \bar{c}_j > U$ then the variable x_j may be permanently fixed to zero.

Since we are not directly solving our problem's LP relaxation, we cannot make use of this basic form; instead, we employ a variant of reduced cost variable fixing based on our solution $\overline{\mathbf{u}}$ to the Lagrangian dual problem. Once $z_{LR}(\overline{\mathbf{u}})$ is satisfactorily close to z_{LD}, we can create a new inequality for determining if a particular variable y_{ij}^t can be fixed to 0 using $z_{LR}(\overline{\mathbf{u}})$ in place of z_{LP}, optimal dual values obtained from solving $z_{SP}(\overline{\mathbf{u}})$ in place of $\bar{c}_j$, and the same upper bound U. The effectiveness of this technique is due in large part to the fact that for our problem $z_{LD} = z_{LP}$, which we stated earlier without proof and will now prove.

Proof that $z_{LD} = z_{LP}$

Theorem 1. If z_{LD} and z_{LP} denote the optimal values for our problem's Lagrangian dual and LP relaxation, respectively, then $z_{LD} = z_{LP}$.

Proof. We proceed by demonstrating that $z_{SP}(\mathbf{u})$ is integral, which allows us to invoke a theorem stated in [18] that will complete the proof. To do this, we reframe $z_{SP}(\mathbf{u})$ as the problem of finding a shortest path from a source to a sink on a time-expanded network $\mathcal{G}_{\mathcal{T}} = (\mathcal{N}_{\mathcal{T}}, \mathcal{A}_{\mathcal{T}})$. The set

$\mathcal{N}_\mathcal{T}$ is comprised of nodes $(i,t) \in \mathcal{N} \times W_i$, and $\mathcal{A}_\mathcal{T}$ is a set of arcs of the form $((i,t),(j, s := \max\{t+\tau_{ij}, e_j\}))$ for $j \in \delta^+(i)$ and $t \in W_{(i,j)}$, each with cost $c_{((i,t),(j,s))} = c_{ij} - u_i$, together with a set of special arcs $((0',t),(0',t+1))$ for $e_{0'} \leq t \leq \ell_{0'} - 1$ (to account for an early arrival time at $0'$), all with cost 0. The source node will be $S = (0,0)$, and the set of zero-cost arcs we included allows for a single sink node $T = (0', \ell_{0'})$. Then if we define sets

$$\delta^-((i,t)) = \{(j,s) \in \mathcal{N}_\mathcal{T} : ((j,s),(i,t)) \in \mathcal{A}_\mathcal{T}\}$$

$$\delta^+((i,t)) = \{(j,s) \in \mathcal{N}_\mathcal{T} : ((i,t),(j,s)) \in \mathcal{A}_\mathcal{T}\}$$

and represent binary arc usage variables with $\mathbf{y}$ we can pose the problem as

$$\begin{aligned}
\textbf{min} \quad & \sum_{(i,t)\in\mathcal{N}_\mathcal{T}\setminus\{T\}} \sum_{(j,s)\in\delta^+(i,t)} (c_{ij} - u_i) y_{((i,t),(j,s))} \\
\textbf{s.t.} \quad & -\sum_{(j,s)\in\delta^+(S)} y_{(S,(j,s))} = -1 && (2.6) \\
& \sum_{(j,s)\in\delta^-(i,t)} y_{((j,s),(i,t))} - \sum_{(j,s)\in\delta^+(i,t)} y_{(i,t),(j,s))} = 0 \qquad \forall (i,t) \in \mathcal{N}_\mathcal{T} \setminus \{S,T\} && (2.7) \\
& \sum_{(j,s)\in\delta^-(T)} y_{((j,s),T)} = 1 && (2.8) \\
& \mathbf{y} \text{ binary,}
\end{aligned}$$

which is a standard shortest path problem (SPP) with no negative cycles (since time must increase when an arc is traversed, node (i,t) cannot be visited more than once), whose extreme points are known to be integral. Thus we may conclude by Theorem 10.3 of [18] and its corollary that $z_{LD} = z_{LP}$. □

Generic Procedure

Now that we have established the integrality of $z_{SP}(\mathbf{u})$ and the strength of z_{LD} as a lower bound, we present a detailed description of our reduced cost variable fixing procedure. To simplify things notationally, we represent the constraints of (2.1) that we move into the objective in our Lagrangian relaxation as $A\mathbf{y} = b$ and the remaining constraints as $C\mathbf{y} = d$. Now we are able to more simply express $z_{LR}(\mathbf{u})$ as

$$\begin{aligned} z_{LR}(\mathbf{u}) \;=\; b^\top \mathbf{u} \quad + \quad & \textbf{min} \quad (c - A^\top \mathbf{u})^\top \mathbf{y} \\ & \textbf{s.t.} \quad C\mathbf{y} = d \\ & \qquad\quad \mathbf{y} \geq \mathbf{0}. \end{aligned}$$

Note that the variables $\mathbf{y}$ are no longer required to be binary due to the integrality of $z_{SP}(\mathbf{u})$. Thus we may take its LP dual, given by

$$\begin{aligned} \textbf{max} \quad & d^\top \mathbf{v} \\ \textbf{s.t.} \quad & C^\top \mathbf{v} \leq c - A^\top \mathbf{u} \\ & \mathbf{v} \text{ unrestricted.} \end{aligned}$$

We see from this dual that the reduced cost for a particular variable y_j is given by $(c - A^\top \mathbf{u} - C^\top \mathbf{v})_j$. All of the constructs are now in place for us to outline our reduced cost variable fixing method, which we present in Theorem 2 below.

Theorem 2. Let $\overline{\mathbf{u}}$ be the vector of Lagrange multipliers returned by our subgradient algorithm, and suppose $\overline{\mathbf{y}}$ and $\overline{\mathbf{v}}$ are the associated optimal solutions to $z_{SP}(\mathbf{u})$ and its LP dual, respectively. If U is an upper bound on the optimal value of the original IP and for some index j we have

$$\overline{y}_j = 0 \;\text{ and }\; z_{LR}(\overline{\mathbf{u}}) + (c - A^\top \overline{\mathbf{u}} - C^\top \overline{\mathbf{v}})_j \geq U$$

then the variable y_j can be fixed to 0 in the original IP.

Proof. Let $z^j_{LR}(\mathbf{u})$ denote the Lagrangian relaxation with the added restriction $y_j = 1$, i.e.

$$\begin{aligned} z^j_{LR}(\mathbf{u}) \;=\; b^\top \mathbf{u} \;\; + \;\; \textbf{min} \quad & (c - A^\top \mathbf{u})^\top \mathbf{y} \\ \textbf{s.t.} \quad & C\mathbf{y} = d \\ & \mathbf{y} \geq \mathbf{0} \\ & e_j^\top \mathbf{y} = 1. \end{aligned}$$

Note that $z^j_{LR}(\mathbf{u})$, its minimization subproblem, provides a lower bound for IP^j, the optimal IP value when y_j is fixed to 1. The LP dual of $z^j_{SP}(\mathbf{u})$ looks like

$$\begin{aligned} \textbf{max} \quad & d^\top \mathbf{v} + \lambda \\ \textbf{s.t.} \quad & (C^\top \mathbf{v})_i \leq (c - A^\top \mathbf{u})_i \qquad \forall \text{ indices } i \neq j \\ & (C^\top \mathbf{v})_j + \lambda \leq (c - A^\top \mathbf{u})_j \\ & \mathbf{v}, \lambda \text{ unrestricted} \end{aligned}$$

If we set $\overline{\lambda} = (c - A^\top \overline{\mathbf{u}} - C^\top \overline{\mathbf{v}})_j$, we see that $(\overline{\mathbf{v}}, \overline{\lambda})$ is feasible for the LP dual of $z^j_{SP}(\overline{\mathbf{u}})$, so that by weak duality we have

$$b^\top \overline{\mathbf{u}} + d^\top \overline{\mathbf{v}} + \overline{\lambda} \leq z^j_{LR}(\overline{\mathbf{u}}) \leq IP^j.$$

We can rewrite the left hand side of this inequality as

$$b^\top \overline{\mathbf{u}} + d^\top \overline{\mathbf{v}} + \overline{\lambda} = b^\top \overline{\mathbf{u}} + (c - A^\top \overline{\mathbf{u}})^\top \overline{\mathbf{y}} + \overline{\lambda} = z_{LR}(\overline{\mathbf{u}}) + (c - A^\top \overline{\mathbf{u}} - C^\top \overline{\mathbf{v}})_j,$$

so that we have

$$U \leq z_{LR}(\overline{\mathbf{u}}) + (c - A^\top \overline{\mathbf{u}} - C^\top \overline{\mathbf{v}})_j \leq z^j_{LR}(\overline{\mathbf{u}}) \leq IP^j,$$

i.e. IP^j is at least as large as our known upper bound U, meaning we can safely fix y_j to 0 in our original IP without eliminating any optimal solutions. □

Constructing a Dual Solution

Our vector **v** of dual variables for the formulation discussed in section 4.2.1 has an entry $v_{(i,t)}$ corresponding to each of its flow balance constraints at intermediary nodes in $\mathcal{N}_\mathcal{T}$ and special entries v_0 and $v_{0'}$ for the source and sink node flow constraints, respectively, and the dual problem has a constraint for each arc $((i,t),(j,s := \max\{e_j, t+\tau_{ij}\}))$ of the form

$$-v_{(i,t)} + v_{(j,s)} \leq c_{ij} - u_i,$$

meaning that each variable y_{ij}^t will have a reduced cost of $c_{ij} - u_i + v_{(i,t)} - v_{(j,s)}$.

Given that we do not use standard LP techniques (e.g. simplex) to solve $z_{SP}(\mathbf{u})$, the question of how to obtain the dual variable values needed to compute reduced costs naturally arises. As it happens, we are able to use labels assigned to nodes by our dynamic programming algorithm for solving SPPTW to construct a feasible solution to its LP dual and subsequently for reduced cost variable fixing. Recall that a node i is given a label $T(P)$ that records the total cost and travel time for each pareto-optimal (i.e. non-dominated) path P from 0 to i.

Suppose that each node i has been assigned n_i labels by our algorithm, which we will hereafter express as $(d_1^i, t_1^i), (d_2^i, t_2^i), \ldots, (d_{n_i}^i, t_{n_i}^i)$ where d and t are used to denote cost (i.e. length) and travel time, respectively. Suppose also that these labels are indexed in such a way that $d_1^i > d_2^i > \cdots > d_{n_i}^i$ and $t_1^i < t_2^i < \cdots < t_{n_i}^i$. Recall that the LP dual of our SPPTW has a constraint for each arc of the form

$$-v_{(i,t)} + v_{(j,s)} \leq c_{ij} - \overline{u}_i.$$

If node i and node j have labels at times t and s, respectively, i.e. $t = t_k^i$ for some k and $s = t_h^j$ for

some h, it must be the case that

$$d_h^j = (\text{optimal cost of reaching } j \text{ at time } s) \leq d_k^i + c_{ij} - \overline{u}_i$$

since the optimal cost of reaching j at time s is at least as small as the optimal cost of reaching i at time t plus the cost of traveling from i to j. Thus $\overline{v}_{(i,t)} = d_k^i$ and $\overline{v}_{(j,s)} = d_h^j$ are feasible in this case.

If node i has a label at time t but node j does not have a label at time s, choose the value h such that $t_h^j < s < t_{h+1}^j$ and note that node j cannot be reached at time s at a cost less than d_h^j since otherwise there would be a label at s. Thus it must once again be that

$$d_h^j \leq (\text{optimal cost to reach } j \text{ at time } s) \leq d_k^i + c_{ij} - \overline{u}_i$$

since $d_k^i + c_{ij} - \overline{u}_i$ is one potential cost of reaching node j at time s, so using the cost of the closest label at j prior to time s for $\overline{v}_{(j,s)}$ and once again setting $\overline{v}_{(i,t)} = d_k^i$ is feasible.

If node j has a label at time s but node i does not have a label at time t, choose the value k such that $t_k^i < t < t_{k+1}^i$ and note that if we depart i for j at time t_k^i we will arrive at j at or before time s (the only case in which we still arrive at s is when $s = e_j$), meaning accrued cost upon arrival must be no less than d_h^j since otherwise there would not be a label at s (or in the case where $s = e_j$ the cost component of the label would be smaller). Thus it must be that

$$d_k^i + c_{ij} - \overline{u}_i \geq (\text{optimal cost to reach } j \text{ at or prior to time } s) \geq d_h^j$$

and we have that using the cost of the closest label at i prior to time t for $\overline{v}_{(i,t)}$ and setting $\overline{v}_{(j,s)} = d_h^j$ is feasible.

If there is neither a label at i for time t nor a label at j for time s, choose values k and h so that

$t_k^i < t < t_{k+1}^i$ and $t_h^j < s < t_{h+1}^j$, and we can combine the reasoning used above to say

$d_h^j \leq$ (optimal cost to reach j at time s) $\leq$ (optimal cost to reach j at or prior to time s) $\leq d_k^i + c_{ij} - \overline{u}_i$

and we have that using the cost of the closest label at i prior to time t for $\overline{v}_{(i,t)}$ and the cost of the closest label at j prior to time s for $\overline{v}_{(j,s)}$ is feasible.

We have now fully constructed a feasible solution to the LP dual of $z_{SP}(\mathbf{u})$. We now proceed to demonstrate the optimality of this solution, thus allowing us to use it in our reduced cost variable fixing procedure.

Optimality of Dual Solution

Once more considering the given vector $\overline{\mathbf{u}}$ of Lagrange multipliers and a corresponding optimal solution $\overline{\mathbf{y}}$ to $z_{SP}(\overline{\mathbf{u}})$ (obtained by solving an SPPRC), we first note that as $z_{SP}(\mathbf{u})$ was proven to be integral in section 4.2.1, the vector $\overline{\mathbf{y}}$ is binary. Those entries that correspond to arcs used in the SPPRC solution path P will take value 1, with all other entries taking value 0. In order to demonstrate the optimality of our constructed dual solution, we must prove (1) that complementary slackness holds and (2) that our dual objective value matches that of the primal value for $\overline{\mathbf{y}}$.

To see that complementary slackness holds, we will argue that for each arc $((i,t),(j,s))$ such that $\overline{y}_{((i,t),(j,s))} = 1$, the corresponding dual constraint

$$-\overline{v}_{(i,t)} + \overline{v}_{(j,s)} \leq c_{ij} - \overline{u}_i$$

is tight. As $\overline{\mathbf{y}}$ is an optimal SPPRC solution, the labeling algorithm must have assigned labels to i and j at times t and s, respectively. The costs in those labels are our dual values $\overline{v}_{(i,t)}$ and $\overline{v}_{(j,s)}$, and since the arc $((i,t),(j,s))$ was used in a shortest path, the latter must exceed the former by the arc's cost of $c_{ij} - \overline{u}_i$. By the nature of the labeling algorithm the difference cannot exceed this amount, and if it was less, it would imply that j could be reached at time s more cheaply by another route,

contradicting P as a shortest path. Thus all constraints which must be tight are indeed satisfied at equality by our dual solution.

To see that primal and dual objective values match, first note that the primal objective value is the total cost of traversing the path P, given by $\sum_{(i,j)\in P}(c_{ij} - \overline{u}_i)$, and that our dual objective function is simply given by $v_T - v_S$. As we have defined them, we will have $\overline{v}_S = 0$ since the label at time 0 for node 0 is trivially $(0, 0)$, and $\overline{v}_T = \sum_{(i,j)\in P}(c_{ij} - \overline{u}_i)$ since P is the shortest path by which $0'$ can be reached by time $\ell_{0'}$. Thus our primal and dual objective values match, meaning that our constructed dual solution is indeed optimal and may be used in the procedure outlined in Section 4.2.2 and made more explicit below.

Our Procedure

Assuming we have computed some upper bound U on z_{IP}, all the components needed for a reduced cost variable fixing procedure are in place; namely, a vector $\overline{\mathbf{u}}$ of Lagrange multipliers and corresponding optimal primal and dual vectors $\overline{\mathbf{y}}$ and $\overline{\mathbf{v}}$ to $z_{SP}(\mathbf{u})$. Before making the procedure explicit, it is important to note here that due to the way in which our dual solution was constructed, many variables will have identical reduced costs. If for some node pair (i, j) there is no label for node i at time t or at time $t + 1$ and no label for node j at time $s_1 := \max\{e_j, t + \tau_{ij}\}$ or at time $s_2 := \max\{e_j, t+1+\tau_{ij}\}$, the variables $\overline{y}_{ij}^t$ and $\overline{y}_{ij}^{t+1}$ will both have reduced cost $c_{ij} - \overline{u}_i + d_k^i - d_h^j$, where the latest labels at i and j prior to times t and s_1 are the k^{th} and h^{th} labels, respectively, with costs d_k^i and d_h^j. Thus the only time points $t \in W_{(i,j)}$ that need to be considered are those where a label can be found either at i or at j. If we suppose that there are K_{ij} labels (d_k^i, t_k^i) at i and H_{ij} labels (d_h^j, t_h^j) at j whose time components fall within $W_{(i,j)}$ and the extension of $W_{(i,j)}$ to j, respectively, we can store these labels in a set L^{ij} of size $K_{ij} + H_{ij}$ and order them according to time from earliest to latest (after adding τ_{ij} to each of the times on the labels for i), giving us a set to iterate over that will determine all reduced costs for all variables $\overline{y}_{ij}^t$, $t \in W_{(i,j)}$. Now we may outline our procedure:

Algorithm 3 Reduced Cost Variable Fixing for Time-Expanded IP (RCF)

for each $i \in \mathcal{N} \setminus \{0'\}$ **do**

for each $j \in \delta^+(i)$ **do**

for each label (d, t) in L^{ij} **do**

If (d, t) is a label for i (for j), identify (d', t'), the latest label at j (at i) prior to time t

if $z_{LR}(\overline{\mathbf{u}}) + c_{ij} - \overline{u}_i + d - d' \geq U$ **then**

Given that $t = t_k^i$ and $t' = t_h^j$ (or vice versa), all variables y_{ij}^r such that $t_k^i \leq r < t_{k+1}^i$ and $t_h^j \leq \max\{e_j, r + \tau_{ij}\} < t_{h+1}^j$ and $\overline{y}_{ij}^r = 0$ may be permanently fixed to 0.

In the computational results that follow, we vary

1. the closeness of $z_{LR}(\overline{\mathbf{u}})$ to the optimal LP value z_{LP} (computed beforehand for proof-of-concept purposes)

2. the closeness of U to the optimal value z_{IP} (known for the instances we tested), and

to get a sense of how convergence of the subgradient algorithm and upper bound quality influence the effectiveness of this technique.

2.5 Computational Results and Conclusions

When we implement the time window pre-processing measures discussed in Section 2.3 followed by the Lagrangian reduced cost variable-fixing technique from Section 2.4 on TSPTW instances from the literature, we are in many cases able to deduce away a sizable portion of potential time points without ever needing to explicitly enumerate all of them. The tables below outline computational results for representative sets of instances from two TSPTW instance libraries in the literature ([20],[8]).

For each instance, we present the gap between its optimal value and the optimal value of its LP relaxation, calculated as $\frac{z_{IP} - z_{LP}}{z_{IP}}$, the number of time points that would be needed to build

its time-indexed IP formulation with no pre-processing, and the numbers of time points needed after applying the pre-processing routine from Section 2.3 followed by various versions of the reduced cost-fixing (RCF) procedure from Section 2.4. The final column of each table is the lowest necessary time point count (achieved when the Lagrangian relaxation value $z_{LR}(\overline{\mathbf{u}})$ is within 0.5% of z_{LP}) following pre-processing as a percentage of the total number of potential time points. We use the shorthand RCF(k, U) to denote the use of our reduced cost-fixing technique with (1) the first value $z_{LR}(\overline{\mathbf{u}})$ in the course of our subgradient algorithm that comes within $k\%$ of z_{LP} and (2) an upper bound U of z_{IP}.

Note that the node- and arc-based pre-processing routines of Section 2.3 tend to perform much better over the instances of [20] than they do over [8]. We attribute this to the relationship between time window widths and travel times—when they are roughly on par with one another, as in [20], this type of pre-processing is quite effective, but when time window widths are much larger than travel times, as in [8], we have much less luck cutting out large chunks of time points.

As one might expect, RCF is most effective when using a tight upper bound for z_{IP} and allowing the subgradient algorithm to push $z_{LR}(\overline{\mathbf{u}})$ extremely close to z_{LP}, but it is worth noting that significant numbers of time points are identified as unnecessary with looser bounds as well. It should also come as no surprise that RCF performs much better on instances with relatively small LP gaps and struggles with instances with an LP gap exceeding 10%.

Instances from Dumas et al. [8]

Table 2.1: Potential vs. actual time points needed to model instance when procedure RCF is used with $U = z_{IP}$.

Instance	LP Gap	Potential TPs	After 2.3.1–3	RCF(k, z_{IP})					% TPs Needed
				k=10	k=5	k=2	k=1	k=0.5	
n60w20.001	3.0%	50896	40221	31678	14049	9784	8082	7207	14.2
n60w40.002	7.4%	93657	82692	67017	59059	40482	35324	33639	35.9
n60w60.001	3.1%	121978	98492	75594	59548	40258	34296	29651	24.3
n60w80.002	16.0%	177060	164149	164149	164077	163990	163863	163679	92.4
n80w20.001	3.5%	95608	79025	56529	39382	27602	22199	19872	20.8
n80w40.001	3.2%	151414	128741	116187	91699	53082	42052	36262	23.9
n80w60.001	10.8%	220908	146699	146699	146278	143890	140501	138802	62.8
n80w80.001	7.9%	267754	200804	198829	186102	167667	158313	152446	56.9
n100w20.001	3.2%	130777	99993	53543	38406	26463	22484	19254	14.7
n100w40.001	4.8%	213381	160918	149358	127884	95225	85603	77800	36.5
n100w60.001	11.1%	354230	327472	327472	321920	313894	310587	308605	87.1

Table 2.2: Potential vs. actual time points needed to model instance when procedure RCF is used with $U = 1.01 * z_{IP}$.

Instance	LP Gap	Potential TPs	After 2.3.1–3	RCF$(k, 1.01z_{IP})$					% TPs Needed
				k=10	k=5	k=2	k=1	k=0.5	
n60w20.001	3.0%	50896	40221	34115	16129	11283	9821	9138	18.0
n60w40.002	7.4%	93657	82692	70653	63373	46925	41475	38241	40.8
n60w60.001	3.1%	121978	98492	78517	65478	46913	41666	38788	31.8
n60w80.002	16.0%	177060	164149	164149	164132	164063	164019	163984	92.6
n80w20.001	3.5%	95608	79025	59318	45630	32913	28643	26357	27.6
n80w40.001	3.2%	151414	128741	119108	102729	69105	56248	48292	31.9
n80w60.001	10.8%	220908	146699	146699	146541	145915	143841	142355	64.4
n80w80.001	7.9%	267754	200804	199488	190727	178440	171822	164786	61.5
n100w20.001	3.2%	130777	99993	58091	43668	31375	27491	24966	19.1
n100w40.001	4.8%	213381	160918	151467	134767	113780	105577	98555	46.2
n100w60.001	11.1%	354230	327472	327472	324829	316248	313369	311773	88.0

Instances from Dumas et al. [8], continued

Table 2.3: Potential vs. actual time points needed to model instance when procedure RCF is used with $U = 1.02 * z_{IP}$.

Instance	LP Gap	Potential TPs	After 2.3.1–3	RCF$(k, 1.02z_{IP})$					% TPs Needed
				k=10	k=5	k=2	k=1	k=0.5	
n60w20.001	3.0%	50896	40221	35935	18975	12913	11382	10777	21.2
n60w40.002	7.4%	93657	82692	72096	67065	53671	46932	44245	47.2
n60w60.001	3.1%	121978	98492	82676	69961	55206	48946	45581	37.4
n60w80.002	16.0%	177060	164149	164149	164148	164123	164081	164063	92.7
n80w20.001	3.5%	95608	79025	64454	49752	37298	33653	32171	33.6
n80w40.001	3.2%	151414	128741	120994	108796	85028	72779	64553	42.6
n80w60.001	10.8%	220908	146699	146699	146672	146221	145920	144363	65.3
n80w80.001	7.9%	267754	200804	200731	194385	183976	180202	175958	65.7
n100w20.001	3.2%	130777	99993	63744	48421	35987	31943	29982	22.9
n100w40.001	4.8%	213381	160918	154509	140897	126206	123026	117413	55.0
n100w60.001	11.1%	354230	327472	327472	325950	318062	315680	314180	88.7

Table 2.4: Potential vs. actual time points needed to model instance when procedure RCF is used with $U = 1.05 * z_{IP}$.

Instance	LP Gap	Potential TPs	After 2.3.1–3	RCF$(k, 1.05z_{IP})$					% TPs Needed
				k=10	k=5	k=2	k=1	k=0.5	
n60w20.001	3.0%	50896	40221	39662	29077	21007	18028	16570	32.6
n60w40.002	7.4%	93657	82692	77355	74513	66516	62712	61202	65.3
n60w60.001	3.1%	121978	98492	92537	79391	72684	70145	67786	55.6
n60w80.002	16.0%	177060	164149	164149	164149	164149	164149	164149	92.7
n80w20.001	3.5%	95608	79025	70160	58679	51464	48931	47766	50.0
n80w40.001	3.2%	151414	128741	127407	118954	111732	107313	104197	68.8
n80w60.001	10.8%	220908	146699	146699	146699	146697	146687	146581	66.4
n80w80.001	7.9%	267754	200804	200804	199973	195539	192350	189552	70.8
n100w20.001	3.2%	130777	99993	78670	63740	51376	48375	46053	35.2
n100w40.001	4.8%	213381	160918	159095	153026	145226	144253	142277	66.7
n100w60.001	11.1%	354230	327472	327472	327466	325957	323254	321310	90.7

Instances from Langevin et al. [20]

Table 2.5: Potential vs. actual time points needed to model instance when procedure RCF is used with $U = z_{IP}$.

Instance	**LP Gap**	**Potential TPs**	**After 2.3.1–3**	**RCF**(k, z_{IP})					**% TPs Needed**
				k=10	**k=5**	**k=2**	**k=1**	**k=0.5**	
N20ft301	0.0%	10668	5001	148	148	88	88	88	0.8
N20ft302	1.2%	10556	7610	1194	591	496	473	430	4.1
N20ft303	0.0%	9717	5441	384	200	200	41	41	0.4
N20ft304	0.0%	10706	5657	156	86	77	77	77	0.7
N20ft305	0.0%	10302	5547	344	122	122	122	122	1.2
N40ft401	0.2%	49055	36218	6270	3305	2306	1774	1601	3.3
N60ft302	0.5%	80456	53748	10164	5898	2997	2129	1759	2.2
N60ft303	0.0%	78734	47978	10179	4985	1948	1723	1459	1.9
N60ft304	0.4%	83769	65564	17206	8470	4132	3061	2657	3.2
N60ft305	1.1%	74983	48364	11904	4451	3031	2364	1928	2.6
N60ft306	1.4%	83581	67434	21603	14431	8366	7076	6052	7.2

Table 2.6: Potential vs. actual time points needed to model instance when procedure RCF is used with $U = 1.01 * z_{IP}$.

Instance	**LP Gap**	**Potential TPs**	**After 2.3.1–3**	**RCF**$(k, 1.01z_{IP})$					**% TPs Needed**
				k=10	**k=5**	**k=2**	**k=1**	**k=0.5**	
N20ft301	0.0%	10668	5001	184	184	93	93	93	0.9
N20ft302	1.2%	10556	7610	1287	733	601	555	526	5.0
N20ft303	0.0%	9717	5441	450	206	206	187	187	1.9
N20ft304	0.0%	10706	5657	217	101	92	80	80	0.7
N20ft305	0.0%	10302	5547	396	141	141	129	129	1.3
N40ft401	0.2%	49055	36218	6935	3913	2952	2472	2225	4.5
N60ft302	0.5%	80456	53748	11508	7296	3829	3096	2728	3.4
N60ft303	0.0%	78734	47978	11160	5902	2799	2531	1992	2.5
N60ft304	0.4%	83769	65564	19005	9986	5797	4671	4022	4.8
N60ft305	1.1%	74983	48364	12843	5459	3698	3189	2691	3.6
N60ft306	1.4%	83581	67434	24146	16761	10477	9126	8051	9.6

Instances from Langevin et al. [20], continued

Table 2.7: Potential vs. actual time points needed to model instance when procedure RCF is used with $U = 1.02 * z_{IP}$.

Instance	**LP Gap**	**Potential TPs**	**After 2.3.1–3**	**RCF**$(k, 1.02z_{IP})$					**% TPs Needed**
				k=10	**k=5**	**k=2**	**k=1**	**k=0.5**	
N20ft301	0.0%	10668	5001	200	200	102	102	102	1.0
N20ft302	1.2%	10556	7610	1333	886	731	712	652	6.2
N20ft303	0.0%	9717	5441	487	248	248	200	200	2.1
N20ft304	0.0%	10706	5657	229	113	113	83	83	0.8
N20ft305	0.0%	10302	5547	518	143	143	136	136	1.3
N40ft401	0.2%	49055	36218	7985	4757	3606	3101	2915	5.9
N60ft302	0.5%	80456	53748	14589	8658	5154	3999	3467	4.3
N60ft303	0.0%	78734	47978	12200	6851	3735	3390	2866	3.6
N60ft304	0.4%	83769	65564	21157	11837	7487	6291	5744	6.9
N60ft305	1.1%	74983	48364	14245	6784	4515	3795	3494	4.7
N60ft306	1.4%	83581	67434	26228	18965	12415	11115	10227	12.2

Table 2.8: Potential vs. actual time points needed to model instance when procedure RCF is used with $U = 1.05 * z_{IP}$.

Instance	**LP Gap**	**Potential TPs**	**After 2.3.1–3**	**RCF**$(k, 1.05z_{IP})$					**% TPs Needed**
				k=10	**k=5**	**k=2**	**k=1**	**k=0.5**	
N20ft301	0.0%	10668	5001	263	263	143	143	143	1.3
N20ft302	1.2%	10556	7610	1603	1095	993	993	984	9.3
N20ft303	0.0%	9717	5441	583	313	313	283	283	2.9
N20ft304	0.0%	10706	5657	336	189	156	132	132	1.2
N20ft305	0.0%	10302	5547	626	244	244	177	177	1.7
N40ft401	0.2%	49055	36218	9950	6745	5748	5294	5118	10.4
N60ft302	0.5%	80456	53748	19562	14143	9317	7776	7344	9.1
N60ft303	0.0%	78734	47978	16588	9954	6358	6104	5648	7.2
N60ft304	0.4%	83769	65564	26141	18964	12308	10789	10448	12.5
N60ft305	1.1%	74983	48364	18911	10964	7797	7050	6474	8.6
N60ft306	1.4%	83581	67434	34839	25786	19286	17937	16670	19.9

CHAPTER 3
A COMPUTATIONAL FRAMEWORK FOR MIXED-INITIATIVE PLANNING OF MANNED SPACEFLIGHT OPERATIONS

3.1 Introduction

In responding to off-nominal situations in space operations, current and past manned spaceflight missions have relied heavily on the support of Mission Control for activity prioritization and re-planning. Utilizing Mission Control in this way makes sense as it has significantly more personnel and expertise at its disposal than does the crew. However, manned spaceflight missions with deep space destinations such as Mars face the challenge of long communication delays between the crew and Earth-based mission control. Communication delays of several minutes or more will negate much of the advantage Mission Control currently has over crew-derived decisions. In these situations, astronauts will need to make decisions about short-term planning autonomously, without this support of Mission Control.

Due to the inherent complexity of space operations, the sparsity of resources, and high work demands, the re-planning of day-to-day activities is a cognitively demanding task, and requires accounting for a wide range of constraints and preferences. As a result, technologies for mixed-initiative planning (MIP) are being developed to support astronauts in completing re-planning tasks. An effective MIP system can be constructed over three basic building blocks: (1) a formal representation of the work that is to be performed, including constraints within the work and resources, (2) algorithms that can reason about the work and aid in or perform re-planning, and (3) an interface that allows human agents to interact with the plan, and ideally builds on a common representation of a plan for the human and the planning system.

The research presented in this chapter focuses on the first two building blocks; we describe a

system consisting of a computational modeling framework that can simulate work together with an optimization algorithm that can rapidly explore the search space associated with the re-planning of activities. The computational modeling framework in use is called Work Models that Compute (WMC) and has previously been used to analyze and synthesize function allocation between air traffic controllers and pilots [1], a pilot and an autoflight system in the flight deck of an aircraft [2], [3], and human and robotic agents in manned space flight operations [4], [5]. The optimization algorithm consists of a generic set of plan optimization heuristics that can be used to generate or modify plans with the goal of reaching optimality or near-optimality with respect to one or more plan objectives. We take inspiration from meta-heuristic and local search concepts commonly used by operations researchers to tackle (machine) scheduling problems. Within the development of these two components, the right coupling between them as well as with the human interface have been critical design considerations.

Subsequent sections provide a brief overview of related work, followed by a discussion of the general planning framework. A more specific implementation is provided in the context of a specific scenario, followed by two case studies demonstrating capabilities of the prototype system.

3.2 Background

Manned spaceflight involves safety-critical and complex operations, in which activities need to be synchronized and resources matched with processes, events and demands in the work environment, under various degrees of uncertainty. Near real-time, crew-centered mixed initiative planning requires a system that can efficiently derive feasible schedules (in several minutes at most) with varying degrees of human involvement in the planning process. As astronauts are not typically experts in mission planning or constraint management, such a system should have automated re-planning capabilities to support them in active constraint identification/resolution and resource management, but should also be able to accept their contextual knowledge and consider their preferences.

Mixed-initiative planning (MIP) relies on optimally combining a human planner's expertise

with technological capabilities, allowing human and automation to jointly come up with plans that are feasible and satisfy a set of predefined objectives [21, 22]. There is a wide variety of examples of this type of planning architecture [23, 24, 25, 26, 21, 22, 27]. Only a limited number of these systems are designed specifically for on-board (re-)planning over a short (24hr) time horizon by astronauts; instead most aim to support mission planners several years, months or weeks before an actual mission (for an overview of existing planning systems for space operations, see [28]).

Nearly all of these planning systems use some form of timeline representation showing the resources and states that are key to the mission, but each uses its own array of technologies and implementation options for automated planning capabilities [28]. Systems range from fully-automated planning capabilities to human manipulation of plans through specialized interfaces. Most of the early planning systems were developed with specialized constraint-checking algorithms, specific to certain missions; more broadly applicable systems have been developed more recently.

Tools used aboard the International Space Station today include Onboard Short Term Plan Viewer (OSTPV) [29] and Playbook [30]. OSTPV relies on communication between mission planners and astronauts for plan updates, while Playbook, though it began largely as a plan visualization tool, now also features some crew re-planning capabilities, e.g. rescheduling of flexible activities and the addition of new activities [31]. This very human-dependent approach to planning uses automation sparingly—to our knowledge, for constraint checking only—which could make re-planning in a complex work environment quite cumbersome for astronauts with a communication delay in deep space.

On the other side of the spectrum, planning algorithms based on artificial intelligence (AI) take a technology-centered approach, but struggle with 'explaining' plan modifications to a human user [32], or fail to consider the complete context in which a plan is deemed feasible or optimal, and therefore require intervention by the human user [22]. In addition, planning research in AI has traditionally decoupled planning of activities with scheduling of resources [25, 27], which has led to challenges in their implementation as in the planning work these two cannot be considered

separately [33].

To bridge the gap between the current automated planning systems and the desired MIP capabilities for astronauts, we argue for a tighter integration of MIP techniques with automated planning capabilities, and comprehensively consider the planning and scheduling problem in the context of manned spaceflight operations. Specifically, we seek a design that is problem-driven (i.e., its underlying representation is built around the needs of autonomous spaceflight planning, rather than attempting to force-fit an established tool or paradigm, and recognizes that activity planning and resource scheduling cannot be de-coupled in this highly-coupled domain), comprehensible (facilitating structures for plans that the human planner can reason about through novel interfaces), computational (providing the domain model suitable for optimization tools), and scale-able (allowing for both detailed and expansive views of the plan).

3.3 Planning Framework

The first part of this work centers around the design of a modeling framework that is both (1) flexible enough to facilitate rapid plan changes and plan optimization, and (2) robust enough to capture the complex constraints and relationships between activities, agents, and resources needed to simulate the work of a given plan.

3.3.1 Framing Planning as an Optimization Problem

For our system to be able to offer new or alternative plans that are "good" with respect to one or more objectives, we need a means of viewing planning from an optimization perspective. Examples of potential planning objectives include:

- minimizing the makespan (time from the start of work to its completion) of the set of tasks that must be completed on a given day, (i.e. efficiency),
- balancing astronaut taskloads in a given time period (i.e. managing fatigue),

- sticking as closely as possible to an original plan in the event of an emergency or unexpected occurrence, (i.e. managing change complexity), and
- aligning the plan with crew preferences or requests.

Improving a plan consisting of dozens (or even hundreds) of tasks spread across multiple crew members with respect to these and other objectives requires a modeling framework that readily lends itself to powerful optimization techniques and algorithms. Fortunately, there exists a fairly natural connection between this planning problem and a well-studied set of problems in the optimization community.

In the context of manned spaceflight, an essential component of any plan is its function allocation—the assignment of responsibility for and/or authority over the execution of tasks to members of the crew and other on-board agents capable of performing work (e.g. autonomous systems or robots). A function allocation together with scheduled start and end times for each task resembles a solution to a machine scheduling problem, a combinatorial optimization problem in which a given set of n jobs must be distributed and sequenced over a set of m machines capable of performing them so as to minimize (or maximize) a chosen objective value (e.g. schedule makespan, job tardiness, weighted sum of job completion times). Numerous variants of this problem have received considerable attention in the optimization, manufacturing systems, and computer science communities (among others) over the last half-century, and many methods for obtaining optimal or near-optimal schedules have been proposed and studied, so that framing this planning process as a variant of the machine scheduling problem is highly desirable. Here, "machines" would be all agents capable of performing work, and our set of "jobs" would consist of all tasks to be completed in a given time frame.

Viewing the planning process through this machine scheduling lens is only possible given a modeling framework robust enough to capture temporal and/or agent-related constraints on individual tasks (e.g. complete Task A no later than time t, Task B must be performed by an agent with

EVA experience), inter-task dynamics (e.g. complete Task D before commencing Task E, Task F is higher priority than Task G), and resource usage and availability constraints (e.g. Task H requires Tool A, Agent 1 has enough oxygen for 2 hours' worth of work). With these considerations in mind, we have developed a framework based on three main high-level constructs: activities, agents, and resources. All members of these three constructs have their own set of attributes that we have divided into three separate categories: characteristics, current state, and plan information. In the operation of our system, characteristics are static input data, current state is dynamic input data, and plan information is output data. A more detailed discussion of each construct and its relevant attributes is given below.

Activities

Activities comprise everything that must be done in a given time frame and are organized in a four-tier structure, with activities (highest level) subdivided into tasks, tasks into subtasks, and subtasks into procedures. This organizational structure helps to facilitate flexibility with respect to the amount of granularity required for planning or re-planning. In the context of planning, we have developed and categorized relevant attributes of activities as follows:

Characteristics Relevant activity characteristics include

- Unique identifier
- Descriptive name
- Earliest (latest) allowed start (completion) time
- Location and duration
- Activity tier, and parent (one tier higher) and child (one tier lower) activities
- Any preceding, succeeding, or parallel activities
- Difficulty level (subjective, rated on a numeric scale)

- Agent skills and/or resources required for execution.

Current State Necessary current state information for an activity includes

- a flag indicating whether or not it is already included in the current plan, and
- an "on-track" flag indicating whether or not the activity will be executed according to plan as things currently stand.

Plan Information Plan information for an activity includes

- the agent(s) performing the activity,
- its start and end times, and
- the identifiers of any resources it is using.

Agents

An agent is any entity capable of performing work; a plan's list of agents may include human crew members, or a variety of robots, e.g. robonauts, robotic arms, Astrobees [34], etc. The attributes of agents that we have deemed relevant in the context of planning are as follows:

Characteristics Relevant agent characteristics include

- Unique identifier/name
- Oxygen and/or energy consumption rates
- Skills (to match certain agents with certain activities)
- Movement speed/range of motion.

Current State Current state information for an agent includes

- agent location,
- resources in possession (e.g. tools for performing maintenance),

- fatigue level, and
- availability (busy/idle).

Plan Information Plan information for a particular agent is simply the set of activities that the agent has been tasked with executing in the plan.

Resources

A resource is an inanimate object or supply which must be utilized to perform certain activities. A resource may be a tool, room/building, replacement part, EVA suit, oxygen tank, power cell, or other such implement. Relevant attributes of resources in the context of planning are as follows:

Characteristics Relevant resource characteristics include

- Unique identifier (e.g. resource name + number)
- Energy consumption rate
- Maximum battery/oxygen capacity (when applicable)
- Storage location.

Current State Current state information for a resource includes

- location,
- times available, and
- oxygen and battery levels when applicable.

Plan Information Plan information for a particular resource is, similar to that of an agent, simply the set of activities that utilize the resource in the plan.

These three major constructs are utilized to varying degrees by both components of our planning system, as well as by the accompanying interface, and allow them to communicate with one another in a shared language.

3.3.2 Computational Work Models

Our modeling framework must be robust enough to facilitate simulation of the work involved in executing a plan. The space domain is characterized by complex dependencies between the work of astronauts, the characteristics and availability of resources, and dynamic and uncertain processes in the work environment, reflected in the many flight rules that mission planners need to account for. For example, extra-vehicular activities in space operations may involve dynamical processes in the work environment (e.g., translation of astronauts, movement of robotic arms, or orbital mechanics) that are critical in the ordering and timing of activities and the resulting feasibility and effectiveness of a plan. These many-to-many mappings in the space domain can translate into emergent effects that are difficult to evaluate from purely static analyses traditionally present in decoupled activity planning and resource scheduling.

To be able to evaluate these effects when assessing plan feasibility, WMC allows for detailed modeling of agents (e.g., human performance modeling), activities (e.g., algorithms capturing how an activity acts on the work environment), and resources (e.g., modeling of location). With a plan as input, these models can be simulated through time, to account for how activities within a plan manipulate the environment, and how the work environment in return influences the appropriate timing of activities and the feasibility of a plan. For example, precedence relationships in traditional activity planning are considered as a linkage between two activities, whereas through simulation the interaction between the activities, modeled as an activity's manipulation of physical resources and information states, can be evaluated directly.

In addition, contrary to timeline representations used in existing planning systems [28], simulation of work models can account for human performance related concerns with the (distributed) multi-agent work common in manned spaceflight operations. Many existing methods for evaluating plans and/or allocations of work in multi-agent systems are primarily focused on performance metrics but fail to consider measures for effective coordination and interaction within a team [35].

Teamwork processes that are necessary to coordinate activities between multiple agents can be a significant factor in a plans feasibility or appropriateness. Simulation of such processes can help astronauts explicitly account for these measures by automatically engendering and logging required activities for verifying progress and commanding/controlling other (robotic or human) agents [36].

To illustrate the more detailed modeling that can be performed in WMC, consider a situation in which an activity is assigned to one astronaut, but another astronaut is responsible for that activity by protocol. This implies that there is a requirement for the responsible astronaut to check-up on the astronaut that is performing the work (this is commonly referred to as the authority-responsibility double-bind [37]), and that the responsible astronaut is not truly free from the activity. The cognitive demands on the responsible agent and communication requirements associated with verifying the correct execution of the activity should not be ignored in the evaluation of a plan. WMC is designed to account for these demands by automatically engendering a monitoring and/or confirmation action for the responsible agent, and logging its execution whenever the original activity is performed [38].

Finally, the simulation of work and its interaction with the work environment allows for the tracking of information requirements for the agents involved in the plan. These requirements depend on the allocation of the activities in the multi-agent team, and when the activities are scheduled in the plan. For example, if two activities assigned to separate astronauts are linked to each other through input and output, there is an implicit requirement for communicating this information from one astronaut to the other. Likewise, when a responsible agent needs to verify an authorized agent's activity, the associated information requirements are also accounted for. Measures of these information requirements can then be used to estimate the communication load inherent to a plan. In much the same way, the simulation tracks any required handovers of physical resources from one agent to the other. This measure is representative for the required physical interaction between agents.

3.3.3 Optimizing a Plan

The optimization component of our system approaches the planning process from a machine scheduling perspective by viewing agents as machines and activities as jobs in need of processing. Given that the problem of optimally scheduling jobs on two or more machines with an objective as simple as minimizing time to completion has been shown to be $\mathcal{NP}$-complete (i.e., computationally intractable for large instances) [39], solution techniques are generally heuristic in nature. Local (or neighborhood) search is one such solution technique that has proven effective in handling large, heavily-constrained problem instances (as long as checking whether or not constraints are satisfied is easy), making it our method of choice. To handle the computational legwork required to run a local search algorithm over a plan spanning multiple agents and dozens of activities, we have written an object-oriented program in C++, currently referred to simply as the Optimizer and based largely on the manipulation of the plan information data outlined earlier in this section. To this point in the Optimizer's development, the optimization process itself has remained relatively uncomplicated. This may change as we graduate to larger and more complicated planning scenarios, but so far a simple neighborhood search method has produced high-quality solutions for reasonably-sized test instances in a relatively small amount of time.

We define the "neighborhood" around a given schedule (plan) to be all schedules that can be obtained from it by removing a single activity and reinserting it elsewhere; consequently we have included "remove" and "insert" functions in the Optimizer to facilitate jumping between neighbors. At the outset of the algorithm, a current schedule and an objective value to be improved/optimized are specified, and then at each iteration a neighbor of the current schedule is selected at random and checked for feasibility (i.e. no temporal or resource-related constraints are violated). Randomness can be tailored to select certain agents and/or activities with higher probability when it is convenient or desirable to do so. If this new schedule is indeed feasible and has an objective value that is at least as good as that of its predecessor, the algorithm replaces the latter with the former, and

then repeats. Otherwise, the current schedule remains 'as is' and a different neighboring schedule is selected in the next iteration. This process continues either for a set number of iterations, or until the schedule's objective value reaches a threshold, depending on the user's preference. The algorithm can also consider multiple objectives via added constraints that prevent jumping to a new schedule if doing so would cause one or more objective values to exceed specified thresholds.

Given that as many as several thousand or more such jumps may be necessary to obtain a near-optimal schedule, examining resource availability, usage, and consumption in depth to determine whether the selected neighboring schedule is feasible at each iteration has the potential to be computationally prohibitive. With this in mind, the Optimizer treats resource-related constraints at a coarse, high level and relies on a separate, more robust evaluation process (i.e., WMC) to identify any problems with the newly optimized schedule that may have gone unnoticed in cursory feasibility checks. We now move to discussing this optimization/evaluation process in greater detail.

3.3.4 Optimizer-Work Models Interaction

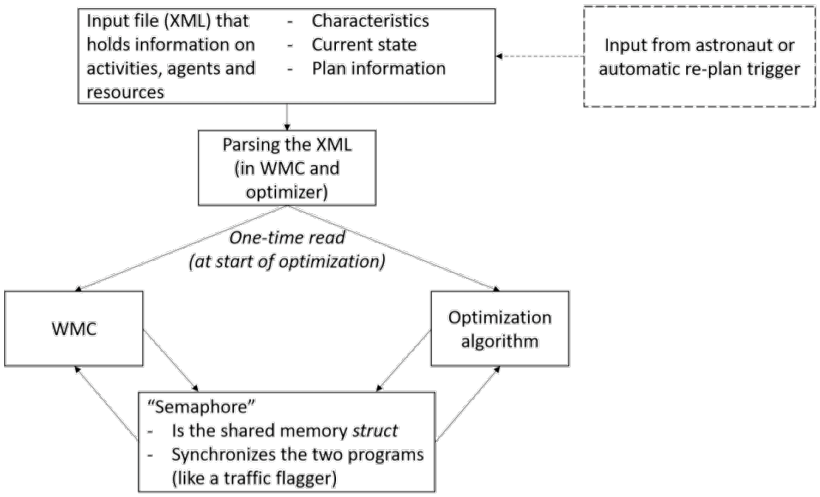


Figure 3.1: Flowchart representation of interaction between the Optimizer and WMC.

Figure 3.1 above provides a high-level depiction of the inner workings of our system. At the outset, the attributes of activities, agents and resources in both the Optimizer and computational

simulation are populated from an external input file that also specifies the constraints and current state of the plan. The Optimizer then iterates on the plan, based on static attributes of the work, agents and resources. Once the Optimizer has converged on a plan that is "good" with respect to one or more planning objectives, it is fed to WMC to evaluate the interactions between activities and work environment, and perform a detailed assessment of its feasibility and optimality in terms of teamwork processes, information requirements and resource handovers. WMC returns values for metrics such as amount of monitoring that is required, the information transfer and physical resource handovers, as well as a list of all activities that had to be delayed to satisfy resource and agent availability constraints that were not accounted for in the Optimizer. The results of this simulation are used to update the constraints and attributes used in the Optimizer, and the Optimizer is run again. The two components iterate until an acceptable plan is found. This arrangement allows for very efficient joint optimization as the majority of the plans are sorted through by the Optimizer and only a smaller set are required to be analyzed in depth by WMC.

To exchange updated information between the two processes during the optimization — plan change(s) to be simulated in WMC and the corresponding metric(s) to be considered in the optimization algorithm — we use a C++ shared memory object ("semaphore"). To limit the required communication between the processes, only the dynamic attributes of objects are communicated through the shared memory; static information is kept only in the local memory. Shared information includes the scheduled time each activity, which agent is assigned to perform each activity, and metrics of interest for the plan.

3.4 Re-planning Using MIP

The general framework discussed above is flexible enough to be used not only as a system for optimizing and evaluating a given plan, but also as a tool for quickly modifying the structure of said plan (e.g., adding or removing activities, changing activity duration, pushing back or moving up activity deadlines) when a crew member or on-board emergency necessitates it.

3.4.1 Workflow

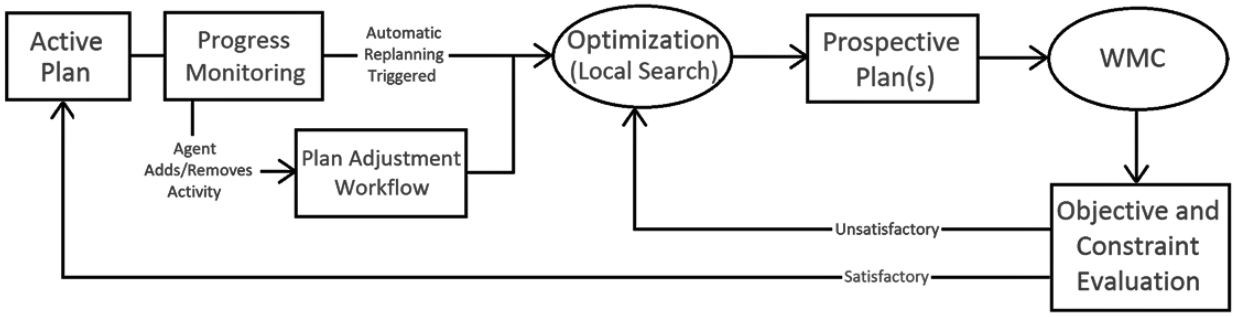


Figure 3.2: Re-planning Workflow

A re-planning operation can be initiated by astronauts, or by automatic detection of an off-nominal situation that affects the current plan. The latter requires automation to track the current plan execution and detect when a situation leads to infeasibility or suboptimality. When an astronaut initiates a re-planning operation, the desired changes with respect to the current plan need to be specified (e.g., insertion of a new activity, changing orders of activities, re-assigning activities, or desired changes in plan attributes such as makespan, required communication etc.). These changes are then reflected in the attributes of the activities, agents and resources in our planning framework, specifically in the characteristics and plan information of each construct.

If a means of accommodating the desired changes isn't immediately apparent within the current schedule, our optimization and simulation processes must be able to assist the crew in adjusting the plan to incorporate them while maintaining feasibility and preferably not straying too far from the original plan. With this consideration in mind, we developed a workflow based on a series of plan adjustment actions (discussed in detail below) to aid in producing an updated feasible schedule.

Following an automatic detection or the aforementioned adjustment process, the system iterates between the Optimizer and WMC, updating plan information until one or more "good" feasible schedules are produced. Ideally multiple options would be presented to the astronaut so that they can then choose the best plan based on their preferences or contextual information not accounted

for in the MIP system. Figure 3.2 above gives a high-level depiction of this workflow.

3.4.2 Plan Adjustment Actions

The plan adjustment actions that we have devised are designed to enable our planning system to modify a given schedule in an orderly and somewhat granular fashion, so that users can visualize or at the very least gauge exactly how much it has been altered from its original state to accommodate a newly introduced activity. These actions are implemented within the Optimizer and can be broadly categorized as either (1) modifications to the existing schedule or (2) modifications to the activity being inserted. Allowable modifications to the existing schedule have been chosen as follows, ordered according to how disruptive to a schedule we perceive them to be, from least disruptive to most:

1. Change the scheduled time of one or more activities, without altering order,
2. Relocate one or more activities within their authorized agents' schedules,
3. Relocate one or more activities across agents,
4. Relax one or more existing activity deadlines,
5. Reduce the duration of one or more activities with lower priority level than the activity to be inserted,
6. Remove one or more lower-priority activities from the schedule.

These actions can be carried out within the Optimizer as many times each as necessary and in any order, giving users flexibility in how they choose to insert an unexpected bit of work. Each time a plan adjustment action is successfully executed, the Optimizer attempts once more to incorporate the new activity using the aforementioned "insert" function. If the insertion is successful, re-planning is complete; if not, another plan adjustment action of the same type is attempted or

the user is prompted for how the system should proceed. In situations where a more automated approach to accommodating a new activity is necessary or desired, a default "plan adjustment workflow", in the form of an ordered list of preferred plan adjustment actions specified either by a user or our by ranking above, will attempt to execute a certain number of plan adjustment actions of each type. The Optimizer will again attempt to insert the new activity after each action before moving to the next type and repeating the process until it is successful. See Figure 3.3 below for a sample automated plan adjustment workflow.

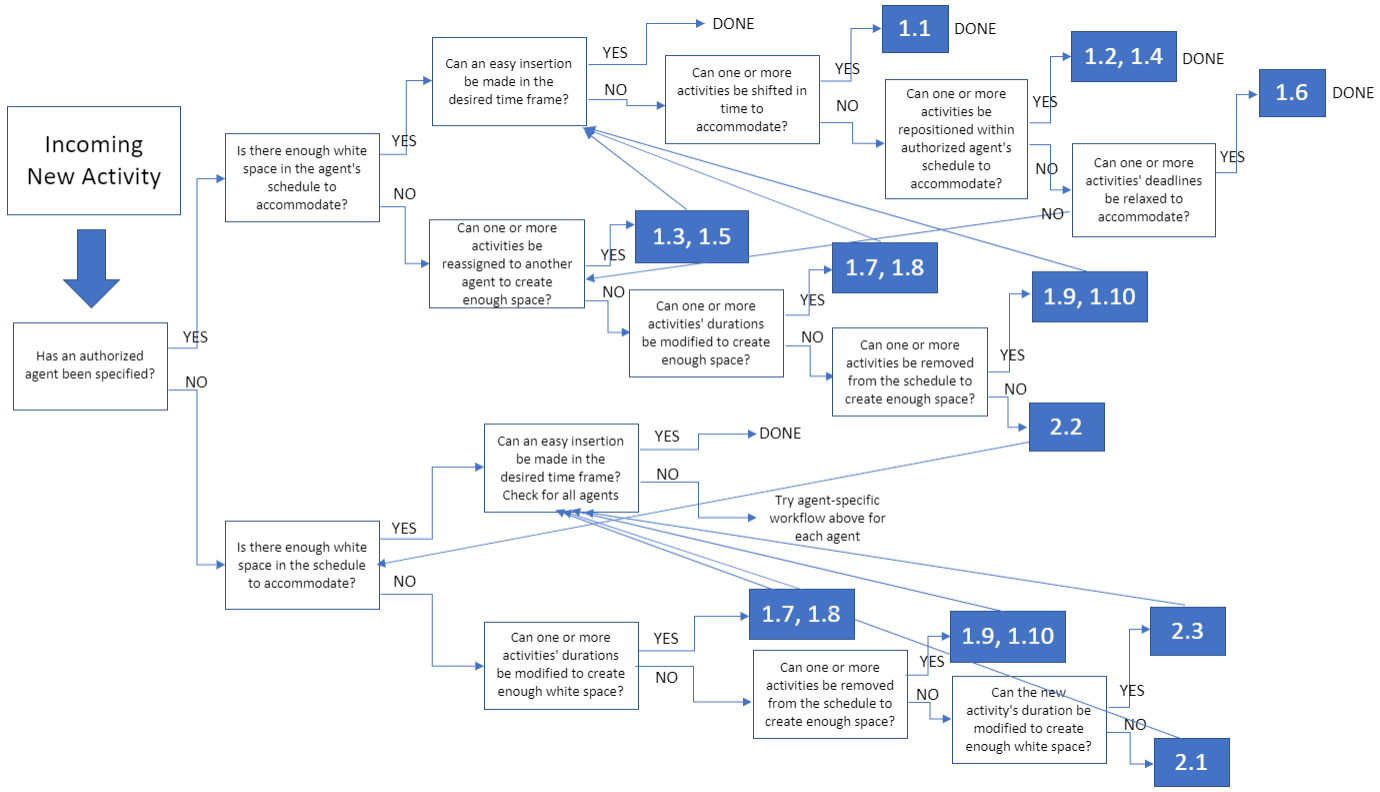


Figure 3.3: Sample Automated Plan Adjustment Workflow

If we find that the existing schedule must be heavily modified to accommodate the new activity and/or it is fairly low priority, we can instead take one or more of the following recovery actions aimed at the new activity itself, ordered according to how disruptive to its execution we perceive them to be and starting from 7 to reflect a continuation of the previous list:

7. Relax the activity's desired time frame or positioning in the schedule,

8. If a required agent has been specified, allow the activity to be assigned to a different agent,

9. Reduce the activity's allocated duration.

3.4.3 Plan Disruption Metric

The granular way in which we have defined our available recovery actions allows us to measure the amount of "disruption" to a plan caused by inserting a new activity into it. As was briefly mentioned in Section 3.3.3, the level to which an optimized or modified plan deviates from the original may be an important metric to keep track of, as crew members will likely need to adjust their workload expectations and preparation routines accordingly. Given the way in which our user-driven and automated "recovery workflows" operate, there is a fairly natural way to keep track, activity-by-activity, of how much a schedule changes in the event that a new activity is added to it.

Tracking how much optimizing or the insertion of a new activity "disrupts" a plan can be done by recording for each activity i in the plan a vector $\mathbf{x}^i$ with entries reflecting changes to the plan:

x_1^i is the magnitude of the change in the activity's scheduled start time (in minutes),

x_2^i is the change in the activity's position relative to other activities in the schedule of the agent assigned to perform the activity, computed as the absolute value of the activity's new sequence position minus its old one (not be applicable if the assigned agent changes, in which case the value is set to zero),

x_3^i is a value between 0 and 1 representing the percentage change (assumed to be a decrease) in the activity's allotted duration,

x_4^i is an indicator taking value 1 if the activity's soft deadline (due date) is violated and 0 otherwise,

x_5^i is an indicator taking value 1 if the activity's assigned agent has changed and 0 otherwise,

x_6^i is an indicator taking value 1 if the activity has been removed from the schedule and 0 otherwise.

By also defining a vector

$$\mathbf{c}^i = [c_1^i\ c_2^i\ c_3^i\ c_4^i\ c_5^i\ c_6^i],$$

of costs (importance levels) and computing $\mathbf{c}^i\mathbf{x}^i$, we can assign to each activity i a "disruption value", so that the disruption value for a schedule S can be expressed as

$$\sum_{i \in S} \mathbf{c}^i\mathbf{x}^i.$$

This plan-wide disruption value is useful both as an objective value for optimization purposes (i.e. making an effort to keep disruption as small as possible) and as an indicator to crew members of how different a proposed schedule will look from their current one. Proper implementation of such a function requires appropriate scaling of the constants comprising each $\mathbf{c}^i$, something that we have not yet addressed but will be one of our focuses moving forward. We now proceed to our case studies, one of which demonstrates how such a plan adjustment workflow may be applied within our planning system.

3.5 Case Studies

The two case studies presented below were conducted over a set of 85 activities scheduled across four human agents (CDR, FE1, FE2, and FE3), representing a full day of NASA's twenty-second NEEMO mission, which ran in 2017. The first demonstrates and compares two different uses of plan adjustment actions to accommodate a new activity, while the second highlights the interaction of optimization with simulation to improve upon an objective—in this case agent workload distribution.

3.5.1 Case Study 1: Inserting a New Activity

Suppose the crew has an additional optional activity that they would like to complete prior to end of day if possible. That is, they would like to insert a new activity, LDRI Color Imagery, into the day's schedule. The activity will take one agent two hours to complete, requires a laptop (physical resource), and ideally would be executed in the early afternoon by agent FE2.

Figure 3.4 below displays the schedule prior to the insertion of the activity (middle subfigure) together with two alternate schedules in which it has been accommodated using different combinations of recovery actions (top and bottom subfigures).

Option 1 (top) requires the following plan adjustments (with disruption values in terms of an activity cost vector **c** included) prior to insertion of the new activity:

- Autonomic Function Data Upload activity reassigned from agent FE2 to agent CDR and scheduled time moved up by 10 minutes ($10c_1 + c_4$).
- Scheduled times for 2 VETTS activities for agent FE2 moved back by 5 minutes ($5c_1$ for each).
- Scheduled times for 2 VETTS activities for agent CDR moved back by 5 minutes to remain in parallel with those for agent FE2 ($5c_1$ for each).
- Scheduled times for activities VETTS T/D and VETTS Q for agent FE2 moved back by 5 minutes ($5c_1$ for each).
- Scheduled time for activity VETTS Q for agent CDR moved back by 5 minutes due to shift of prior VETTS activities ($5c_1$).

This option sees the new activity scheduled in the early afternoon with the desired agent (FE2). Option 2 (bottom) requires the following plan adjustments prior to insertion of the new activity:

- Activity RAPSAP 0 reassigned from agent FE1 to agent FE2 and scheduled time moved up by 20 minutes ($20c_1 + c_4$).
- Scheduled time for activity Mini DNA Q moved up by 420 minutes to avoid a resource conflict (laptop usage), re-ordering activities in agent FE3's schedule ($420c_1 + 11c_2$).
- Scheduled times for activities RAPSAP 0 and AR Sani Tank Purge for agent FE2 moved back by 10 minutes due to resource conflict (laptop usage) created by relocating activity Mini DNA Q ($10c_1$ for each).
- New activity scheduled with agent FE1 rather than preferred agent FE2 (c_4).

This option also sees the new activity scheduled in the early afternoon, but not with the preferred agent.

The sheer number of options for how to include a new activity that will typically be available is apparent even in this very small example. It is therefore of some importance to keep track of how much each proposed schedule differs from the original plan (which can be done using our "disruption function"), as well as how each measures up in terms of other objectives such as makespan, workload balance, and crew satisfaction. Currently, our idea for a more manual plan adjustment process is to have a user create one or more "recovered" schedules within the constraints of the plan using their preferred assortment(s) of plan adjustment actions; when a more automated process is necessary, the system will run through several variations of the same ordering of plan adjustment actions (e.g. each run allowing a different number of each plan adjustment action to be attempted before moving on) and consider the resulting schedules. These schedules can of course be further optimized with respect to one or more of these objectives after the plan adjustment process is complete using our proposed local search algorithm, so that ideally at the end of the plan adjustment process agents are presented with several good new schedules from which to choose.

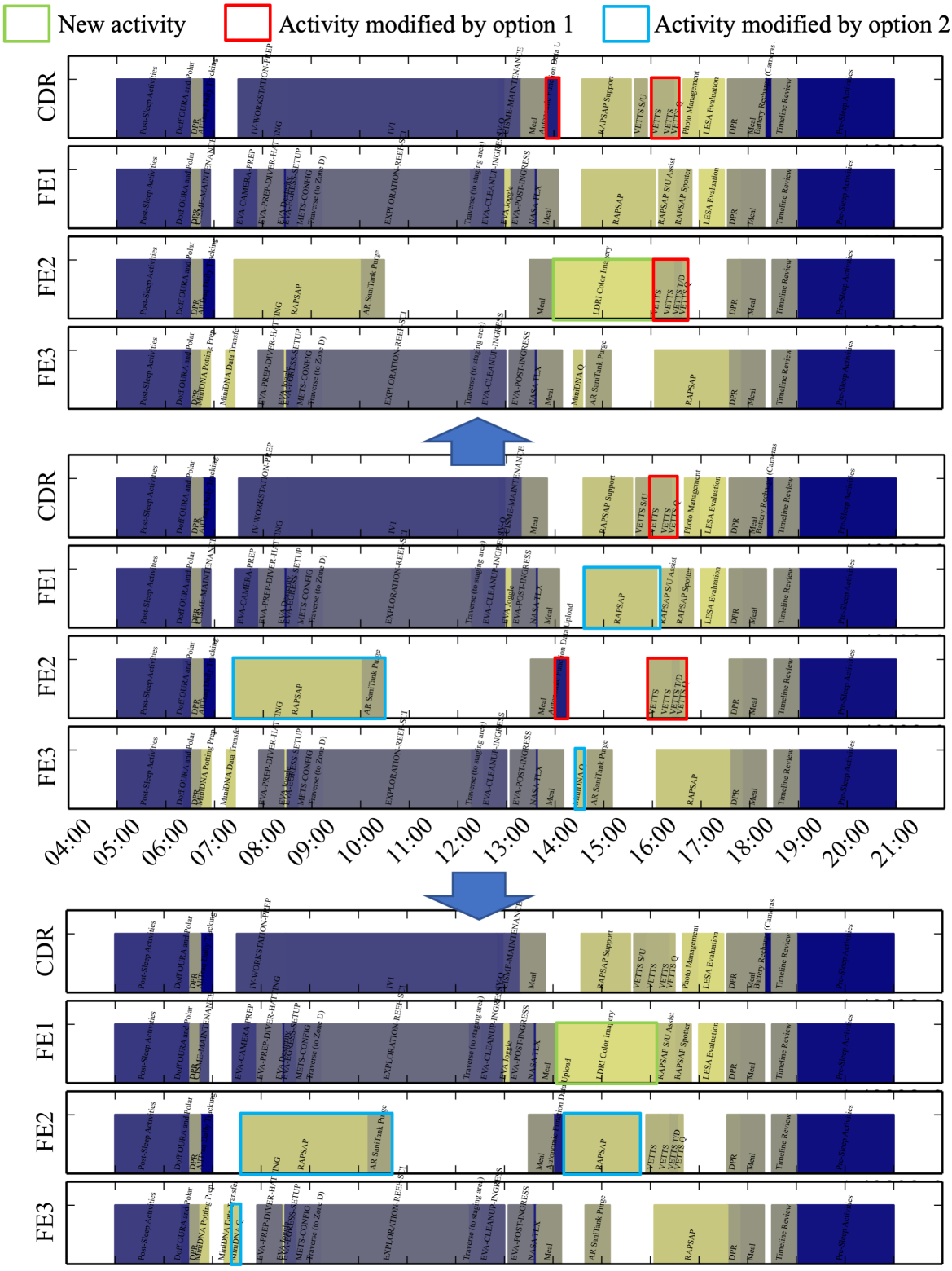


Figure 3.4: Case Study 1: Two different sequences of plan adjustment actions used to insert a new activity into the schedule. Middle image depicts schedule prior to insertion, top and bottom images depict options 1 and 2 for insertion.

3.5.2 Case Study 2: Balancing Agent Workloads

In this second case study, we focus on optimizing the schedule with respect to a particular objective—workload distribution among agents—and in doing so highlight the interaction between optimization and simulation within our computational framework.

We measure workload balance using an agent fatigue estimation model within WMC; it uses the status of each agent and of the work environment to estimate agent fatigue levels at five minute intervals throughout its forward simulation of the plan. The smaller the spread between the highest and lowest estimated fatigue levels at the completion of the day's work, the better the schedule's workload balance. WMC's ability to model and track changes in the work environment during forward simulation means that these fatigue checks can be as simple or as detailed as a user would like them to be. In this case study, we estimate agent fatigue using a simple model of agent circadian rhythm together with assigned physical and mental difficulty ratings for activities. However, given WMC's capacity for modeling work and the work environment in depth, we could easily transition into using more involved work environment-based methods for measuring agent fatigue, e.g. Bayesian networks [40].

To optimize the plan with respect to agent workload distribution, our planning system iterates between the Optimizer and WMC, with the Optimizer passing what it believes is a "good" feasible schedule to WMC for a detailed analysis, and WMC responding by passing back final estimated agent fatigue levels along with any activity delays or resource conflicts not recognized during the stripped-down plan optimization process.

For this case study, running a schedule through the Optimizer involves attempting 5,000 activity removals and re-insertions, sticking with a removal/re-insertion if it leads to an improvement (or no change) in its crude estimation (using activity fatigue ratings) of the gap between the highest and lowest agent fatigue levels and reverting to the previous schedule otherwise.

After an initial run through this optimization process in which agents for activity removal and

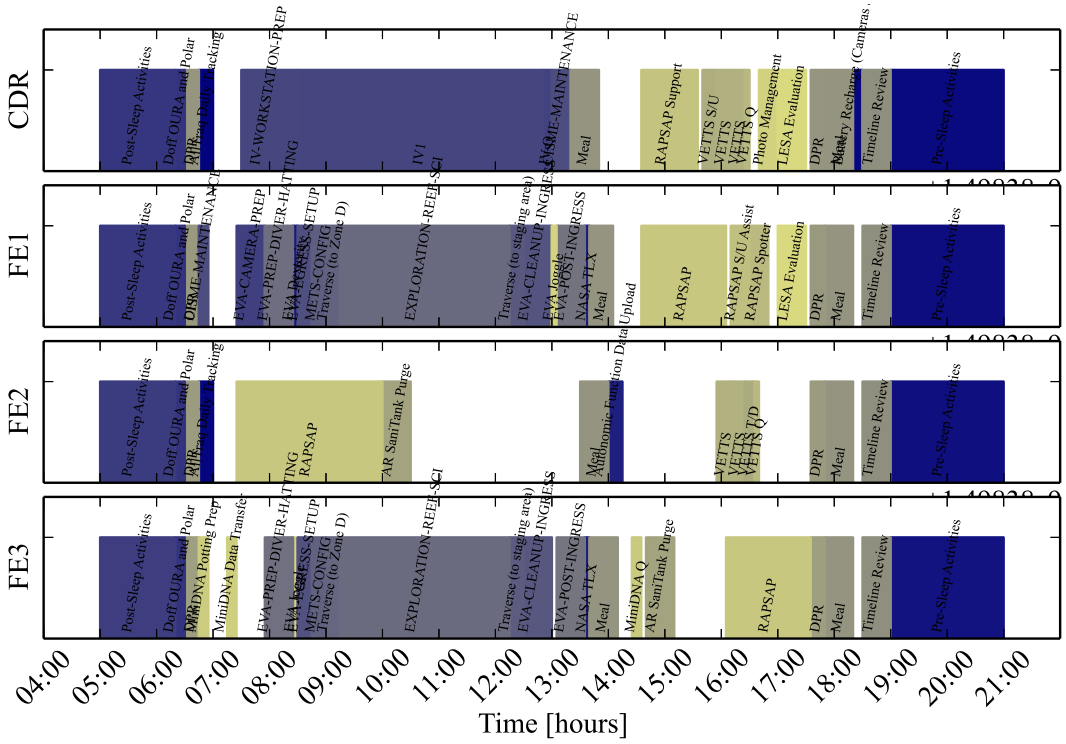
re-insertion are selected uniformly at random, the Optimizer passes it to WMC for simulation. Following simulation, WMC responds with feedback (estimated final fatigue levels and any unforeseen issues) for the Optimizer. The Optimizer then updates its agent selection distribution to reflect these fatigue estimates and attempts to clean up any issues before proceeding through another round of 5,000 attempted removals and re-insertions.

This back-and-forth process continues for six full Optimizer-WMC iterations, terminating with a feasible schedule that looks rather different but boasts a much smaller gap between the highest and lowest final agent fatigue estimates. Figure 3.5 on the page that follows depicts the schedule before and after optimization, and Figure 3.6 on the next page plots WMC's agent fatigue estimates over time for both schedules.

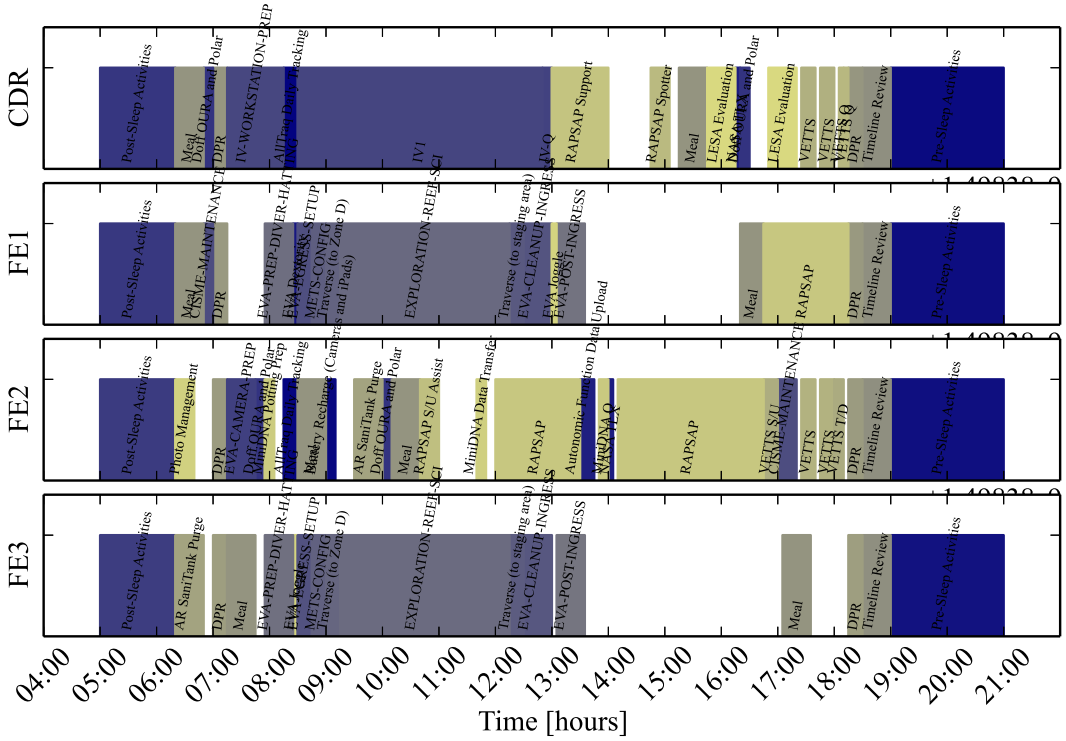
3.6 Conclusions and Future Work

In this chapter, we have outlined the computational framework and tools behind a mixed-initiative planning system for human spaceflight that we have prototyped. Our approach differentiates itself from other MIP systems through its coupling of plan optimization techniques originating in the field of operations research with a detailed work modeling process with its roots in cognitive engineering. The result is a semi-autonomous planning system that works with human agents to produce high-quality, high-fidelity schedules, even in off-nominal or unanticipated conditions. The two case studies presented highlight our planning system's abilities to respond to new or off-nominal circumstances and to optimize plans with respect to a single objective using a combination of local search and simulation.

Moving forward, problems of interest include the development of robust automated plan adjustment workflows like the one depicted in Figure 3.3, optimizing with respect to multiple objective criteria simultaneously, and finding structured ways to accommodate agent preferences within our modeling frameworks.

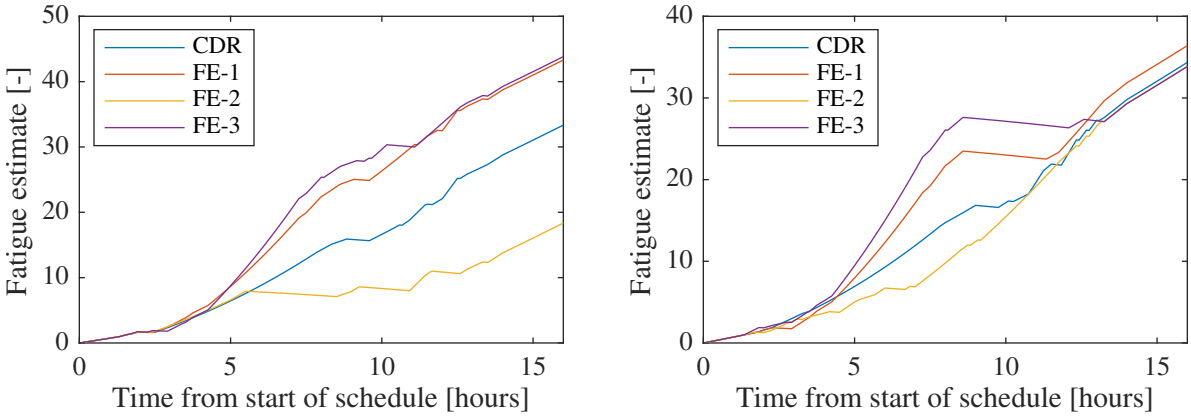


(a) Before optimization



(b) After optimization

Figure 3.5: Case Study 2: Optimization of agent workload distribution. Upper schedule is prior to optimization, lower is after.



(a) Before optimization

(b) After optimization

Figure 3.6: Case Study 2: Optimization of agent workload distribution. The two line graphs at the depict agent fatigue estimates over time, taken every 5 minutes during WMC simulation.

CHAPTER 4
SINGLE MACHINE SCHEDULING WITH TIME- AND JOB-DEPENDENT PROCESSING TIMES AND MAINTENANCE

4.1 Introduction

Machine scheduling problems in which job processing times depend on their start times have interested operations researchers and computer scientists for more than 30 years. These problems are interesting not only from a theoretical perspective, but also from a practical one—varying a job's processing time according to its start time can simulate wear-and-tear on a machine (i.e. deterioration) or a machine operator's increased efficiency as they gain experience in processing certain types of jobs (i.e. learning). Much of the scholarly attention to this point has focused on variants of a linear model for time-dependent deterioration (learning) in which a job j that starts processing t time units after the start of the schedule takes

$$p_j(t) := p_j + a_j t \tag{4.1}$$

time units to process, where p_j is the job's base processing time and a_j is a deterioration or learning coefficient, depending on its sign. This chapter focuses on scheduling jobs with processing times of the form (4.1) on a single machine in a strictly deteriorating setting (i.e. $a_j > 0$ for all jobs j), meaning the later any job starts relative to the start of the schedule, the longer it will take to process.

As optimal algorithms or hardness proofs have been offered for most common scheduling objectives (e.g. makespan, tardiness, sum of weighted completion times) in this particular setting, more recent work has added another wrinkle to the problem by introducing "maintenance periods"

(MPs) of fixed or varying length that can be included in the schedule to mitigate machine deterioration by either partially or fully "resetting the clock"; that is, start times for jobs scheduled after a maintenance period will have a different (more recent) reference point than the start of the schedule. In the sections that follow, we will see that if our aim is to minimize schedule makespan (i.e. time to completion), the optional inclusion of MPs adds an additional layer of complexity to an otherwise easy-to-solve problem.

Our choice to allow for variation of both base processing times and deterioration rates among jobs *and* inclusion of maintenance periods in the schedule stem from our experiences in the human spaceflight scheduling work of Chapter 3, specifically vis-à-vis the management of astronaut fatigue. To accurately depict an astronaut's set of assigned activities in a machine scheduling context, we need to be able to model variation in both the amount of time *and* the physical/mental effort level required to complete an activity. A spacewalk not only takes significantly longer to complete than a check-in with mission control (i.e. $p_{SW} > p_{CI}$), but also requires significantly more mental focus and physical exertion, so that the start time of the spacewalk relative to the beginning of the astronaut's day (read: astronaut's fatigue level) should have more of a negative impact on its "processing time" than that of the check-in (i.e. $a_{SW} > a_{CI}$). Similarly, a chemistry experiment involving a volatile substance may not take as much time as uploading the resulting data to servers on Earth, but it is almost certainly more mentally and physically demanding, so setting $p_{CE} < p_{UD}$ and $a_{CE} > a_{UD}$ would be appropriate.

Rest is another important piece of the calculus of astronaut fatigue. If an astronaut's day started at 7 AM, they likely will have had at least one break or period of reduced activity prior to the start of an important repair job at 5 PM. The inclusion of MPs in the schedule gives us a way to account for these "off" times that could positively impact the astronaut's fatigue and efficiency levels. A similar metaphor can be drawn for machines—down time for repairs and maintenance can greatly improve (or completely restore) initial operating conditions, so that halting processing temporarily is quickly offset by the time saved thanks to increased efficiency.

In this work, we consider the problem of scheduling n jobs and $K < n$ MPs on a single machine so as to minimize makespan. We assume that all MPs have a common fixed length $\ell > 0$ and fully "reset the clock", i.e. start times for jobs are set relative to the end time of the nearest preceding MP rather than the start time of the schedule. From this point forward, we will use standard machine scheduling shorthand in referring to this problem as $1\big|p_j(t) = p_j + a_j t, MP(K)\big|C_{max}$. The left-most entry indicates the number of available machines, the middle entry lists any special job and/or problem characteristics, and the right-most entry specifies the chosen objective ($C_{max} \equiv$ makespan). To our knowledge, the only work that has been published on minimizing makespan when job processing times are defined exactly as above and fixed-length MPs are available for scheduling is that of Wang et al. [41], who use Dantzig-Wolfe decomposition and a branch-and-price algorithm to solve the problem exactly.

4.2 Background and Special Cases

We begin this chapter with a review of known solution methods for special cases of our problem of interest—starting with a result for processing times of the form (4.1) without MPs available for scheduling; following this with a result for optimal placement of a single available MP when processing times are both start time- and position-dependent; and finishing with a result for processing times of the form (4.1) with $a_j = a$ for all jobs j and K available MPs.

4.2.1 Makespan-optimal ordering without MPs [42]

A 2004 survey from Cheng et al. [43] gives a comprehensive overview of the work that has been done on scheduling problems with time-dependent processing times and no available MPs. Among the results presented is that of Gupta and Gupta [42] regarding the makespan-optimal ordering of jobs for the problem $1\big|p_j(t) = p_j + a_j t\big|C_{max}$ with strictly positive deterioration rates a_j. Theorem 2 in that work establishes that scheduling jobs in ascending order of the ratio of their base processing time to their deterioration rate (i.e. $h_j := \frac{p_j}{a_j}$) is makespan-optimal, meaning that the problem is

solved by simply sorting jobs prior to scheduling, which can be done in time $\mathcal{O}(n \log n)$.

4.2.2 Makespan-optimal placement of a single MP with position-dependent deterioration [44]

A significant result involving deteriorating processing times and maintenance belongs to Lodree and Geiger [44], who demonstrated that if job processing times are strictly position- and start-time-dependent, the optimal positioning for a single MP of fixed length ℓ can, under certain assumptions, be easily determined. In their problem setting, if the r^{th} scheduled job starts processing at time $t_{[r]}$, its processing time is given by

$$p_{[r]} := a_{[r]} t_{[r]},$$

where $a_{[r]} \geq 1$ is the deterioration rate associated with position r. They also assume that the schedule starts at time 1 (necessary to escape time 0) and that the MP fully restores processing conditions, so that if the MP is placed in position k (or immediately following the job in the $(k-1)^{th}$ position), job processing times can be expressed as

$$p_{[r]} = \begin{cases} a_{[r]} t_{[r]}, & r = 1, \ldots, k-1 \\ p_{[r-k]}, & r = k+1, \ldots, n+1. \end{cases}$$

They proceed to deduce that schedule makespan when the MP is scheduled in position k can be computed as

$$C_{max}(k) := \ell + \prod_{i=1}^{n-k+1} (1 + a_{[i]}) + \prod_{i=1}^{k-1} (1 + a_{[i]}),$$

and offer the following theorem on optimal MP placement:

Theorem. *The optimal policy for scheduling an MP of length ℓ under position-specific simple linear deterioration with $a_{[r]} \geq 1 \ \forall r$ is as follows: If n is an even integer and $\ell < \prod_{i=1}^{n}(1 + a_{[i]}) - 2\prod_{i=1}^{\frac{n}{2}}(1 + a_{[i]})$, assign the MP to sequence position $k^* = \frac{n}{2} + 1$. If n is odd and $\ell < \prod_{i=1}^{n}(1 + a_{[i]}) - \prod_{i=1}^{\frac{n-1}{2}}(1 + a_{[i]}) - \prod_{i=1}^{\frac{n+1}{2}}(1 + a_{[i]})$, assign the MP to either sequence position*

$k^* = \frac{n+1}{2}$ *or* $k^* = \frac{n+1}{2} + 1$. *Otherwise, do not schedule the MP.*

In short, the makespan-optimal positioning for the MP is as close to the middle of the schedule as possible. This logic will generally continue to be sound for problems with a single MP and processing times of the form (4.1), but certain (unfortunate) relationships between p_j and a_j values will lead to optimal schedules with, for example, three jobs scheduled prior to the MP and one job following it.

4.2.3 The problem with no base processing times and one MP is NP-Hard [45]

Ji et al. [45] proved that when processing times have the form $p_j(t) = a_j t$, absent of any base processing time, and there is one available MP of fixed length ℓ scheduled in position $k < n$, the problem of minimizing schedule makespan is NP-Hard. If we assume the first job kicks off at time 1 (in order to "escape" time 0), schedule makespan can be expressed as

$$\prod_{i=1}^{k}(1 + a_{[i]}) + \ell + \prod_{i=k+1}^{n}(1 + a_{[i]}).$$

If we re-label the first term of the sum as B and let $A = \prod_{i=1}^{n}(1 + a_{[i]})$, the makespan function can be re-written as $B + \frac{A}{B} + \ell$, and differentiating this with respect to B leads us to conclude that makespan is minimized when $B = \sqrt{A}$, which is equivalent to solving an instance of the NP-Hard Product Partition (PP) problem.

4.2.4 Makespan-optimal ordering with common deterioration rate a and K available MPs [46]

Rustogi and Strusevich [46] developed an optimal algorithm for minimizing makespan with processing times of the form (4.1) with $a_j = a$ for all jobs j, as well as an extension of this algorithm to handle MP-dependent deterioration rates. In the setting with $a_j = a$ for all jobs j, the total

processing time for a sequence of k consecutively scheduled jobs can be computed as

$$\sum_{i=1}^{k}(1+a)^{k-i}p_{[i]},$$

where $p_{[i]}$ is the processing time of the i^{th} job in the sequence. Thus the makespan C_{max} for a given sequencing of jobs with a single MP of length ℓ scheduled immediately following the k^{th} job in the sequence is given by

$$C_{max}(1) = \sum_{i=k+1}^{n}(1+a)^{n-i}p_{[i]} + \sum_{i=1}^{k}(1+a)^{k-i}p_{[i]} + \ell.$$

Note that p_j appears in this equation exactly once for each job j, and that if job j is scheduled in position i we have

$$\text{multiplier for } p_j = \begin{cases} (1+a)^{k-i} & \text{if } i \leq k \\ (1+a)^{n-i} & \text{if } i > k. \end{cases}$$

If we assume that the MP will be included in the schedule, the problem of minimizing makespan is now simply that of finding a minimal pairing of p_j values with these multipliers. As we would like the job with the largest p_j to have the smallest multiplier possible in our C_{max} calculation, we start by scheduling that job in the last position on either side of the MP; either side will do since in both cases its multiplier will be 1.

We would then like the job with the second-largest processing time to have as small a multiplier as possible given the positioning of the first job, so we schedule it in the last position on the opposite side of the MP from the job with the largest processing time, where it will also have a multiplier of 1.

Next we consider the job with the third-largest processing time, and again trying to minimize its multiplier we schedule it in the second-to-last position on either side of the MP, since in either case its multiplier is $(a+1)$. The job with the fourth-largest processing time is then placed in

whichever position we didn't choose for the previous job, where it also has multiplier $(a+1)$, and this process continues until all jobs have been placed.

The end result is a makespan-optimal schedule with the MP sandwiched in the middle—immediately following the $\frac{n}{2}^{th}$ job in the sequence in the case where n is even, or immediately following either job $\frac{n-1}{2}$ or $\frac{n+1}{2}$ in the case where n is odd, depending on where we decided to put the job with the smallest processing time.

As to the question of whether or not the MP should be scheduled, if we assume n is even and index jobs from 1 to n in ascending order of their p_j value, one potentially optimal schedule (of many) that can be obtained via this process is $(1, 3, 5, \ldots, n-1, \text{MP}, 2, 4, 6, \ldots, n)$. To determine if the MP is worth scheduling, we need only to calculate

$$C_{max}(0) - C_{max}(1) = \sum_{i=1}^{n}(1+a)^{n-i}p_i - \left(\sum_{i=1}^{\frac{n}{2}}(1+a)^{\frac{n}{2}-i}p_{2i} + \sum_{i=1}^{\frac{n}{2}}(1+a)^{\frac{n}{2}-i}p_{2i-1} + \ell\right).$$

If this difference is non-negative, we schedule the MP; otherwise we do not. A similar comparison can be made for the case where n is odd.

When $K > 1$ MPs of length ℓ are available for scheduling, a natural extension of the algorithm above (continuing to pair largest processing times with smallest multipliers, just with more options each time) again provides an easy way to minimize makespan when we choose to include a fixed number $k' \leq k$ of MPs in the schedule.

Deciding on the optimal number of MPs to include is still easy, as it only requires repeating this simple algorithm up to K times. We would first need to check whether $C_{max}(K-1) - C_{max}(K) < 0$, and if so would then need to check if $C_{max}(K-2) - C_{max}(K-1) < 0$, and so on until a non-negative difference is discovered.

If we further extend the problem by allowing job deterioration rates to vary based on the nearest previously scheduled MP, i.e. job j will have deterioration rate $a^{[x]}$ when scheduled following the x^{th} MP but prior to the $(x+1)^{th}$, we can still determine the makespan-optimal schedule for a set

number of MPs with the same pairing algorithm, and need only to make a few more comparisons than before to determine the optimal number of MPs to include. Instead of terminating when we find a $k \leq K$ such that $C_{max}(k-1) - C_{max}(k) < 0$, we would need to compare all $K+1$ values of $C_{max}(k)$ in case a certain combination of MP-dependent deterioration rates for a small number of MPs would lead to a shorter minimum makespan than that for a larger number.

4.2.5 A branch-and-price algorithm for solving the general problem [41]

Wang et al. [41] have recently proposed a branch-and-price algorithm for solving $1\big|p_j(t) = p_j + a_j t, MP(K)\big|C_{max}$. They use a set-partitioning formulation similar to the one we present in Section 4 as their restricted master problem, and use a separate mixed integer linear programming formulation comprised of variables for job start times and job ordering to obtain a pricing problem for column generation. The algorithm branches on both the number of MPs allowed to be scheduled and on whether or not two jobs are scheduled consecutively.

This is the only work that we have seen that addresses our problem of interest. Given that their algorithm manages to solve small to medium size instances but takes a substantial amount of time and computing power to do so, we will instead focus on alternative (heuristic) methods for obtaining high-quality solutions quickly.

4.3 Problem Statement, Notation, and Characteristics

Suppose we are given a set J of n jobs with processing times of the form (4.1) to be processed on a single machine, along with $K < n$ available MPs, all of length $\ell > 0$. Suppose also that when an MP is scheduled, its end time becomes the reference point (time 0) for the start times of all jobs scheduled after it but before the next scheduled MP. In its most general form, our problem of interest is twofold: we must

1. determine the optimal number $k^* \leq K$ of MPs to include in the schedule, and

2. determine the makespan-optimal ordering of the n jobs and k^* MPs on the machine.

Outside of a brief discussion on determining whether or not to include an MP in the schedule when $K = 1$, this chapter will focus primarily on the second point—assuming we have chosen to include some fixed number of MPs and endeavoring to arrange jobs optimally around them.

We now move to introduce some formulae and notation that enable us to discuss the problem in more precise terms.

4.3.1 Schedule makespan as a function of p_j and a_j values

Given a sequence $(j_1, j_2, \ldots, j_m)$ of jobs from J, the time needed to process them consecutively without maintenance can be expressed as

$$p(j_1, j_2, \ldots, j_m) := p_{j_m} + \sum_{i=1}^{m-1} \left(\prod_{k=i+1}^{m} (1 + a_{j_k}) \right) p_{j_i}.$$

Furthermore, given an arbitrary subset of jobs $S \subseteq J$, the result from (Gupta and Gupta) discussed in Section 4.2 tells us that a makespan-optimal sequence of its elements has them ordered from lowest $h_j = \frac{p_j}{a_j}$ value to highest, with ties broken arbitrarily. To keep notation simple, we will use the relation $j \prec k$ to indicate that job j precedes job k in such a sequence, and will denote the minimal uninterrupted processing time of a set of jobs S as

$$p(S) := p_{j^*} + \sum_{j \in S \setminus j^*} \left(\prod_{\substack{k \in S \\ j \prec k}} (1 + a_k) \right) p_j, \tag{4.2}$$

where j^* is the index of the job in S with the highest h_j value. In light of our assumption that an MP completely "resets the clock", the decision to include some number $k \leq K$ MPs in the schedule effectively partitions J into $k + 1$ subsets $S_1, S_2, \ldots, S_{k+1}$ whose processing times may be computed independently of one another. We can therefore express the makespan for such a

schedule as

$$C(S_1, S_2, \ldots, S_{k+1}) := \sum_{i=1}^{k+1} p(S_i) + k\ell. \tag{4.3}$$

Note that different allocations of jobs to the subsets of the partition will lead to different makespans, dependent on the interplay of p_j and a_j values within subsets.

Now, if $k^* \leq K$ is the optimal number of MPs to include in the schedule and $S_1^*, S_2^*, \ldots, S_{k^*+1}^*$ is the corresponding makespan-optimal partition of J, we can express the optimal schedule makespan C_{max}^* as

$$C_{max}^* := C(S_1^*, S_2^*, \ldots, S_{k^*}^*) = \sum_{i=1}^{k^*+1} p(S_i^*) + k^*\ell.$$

Expressing processing times for sets of jobs and schedule makespan as above allows us to make the following observations regarding the structure of the functions p and C.

4.3.2 The function $p(S)$ is supermodular, but makespan is not

Many set functions of interest in the realms of economics, game theory, computer science, and optimization are known to be supermodular; in words, this means that the marginal cost of of adding a single object to a set of objects grows as the size of the set increases. Of the several equivalent mathematical definitions for supermodularity, we will use the one given below to establish that function p is supermodular.

Definition 1. *A set function $f : U \to \mathbb{R}$ is **supermodular** if, given any pair of subsets $A, B \subseteq U$ with $A \subseteq B$ and any element $e \in U \setminus B$, the below relation holds:*

$$f(A + e) - f(A) \leq f(B + e) - f(B).$$

To demonstrate that p is supermodular according to this definition, we first introduce the notion of an individual job's *contribution* to a set's processing time; that is, by how much the set's processing time would increase (decrease) if the job were added to (removed from) it. Given a set

of jobs S, the contribution of a job $j \in J$ to the processing time $p(S)$ can be computed as

$$contr(j;S) := \left(\prod_{k \in \overline{S}_j} (1 + a_k)\right)\left(p_j + a_j p(\underline{S}_j)\right),$$

where $\underline{S}_j$ and $\overline{S}_j$ are the sets of jobs in S that would precede and succeed, respectively, j in a makespan-optimal sequencing. The value $contr(j;S)$ is equivalent to either $p(S) - p(S-j)$ or $p(S+j) - p(S)$, depending on whether or not j is a member of S. Now we are in a position to prove the supermodularity of p.

Observation 1. *The function p is supermodular.*

Proof. Given subsets $A, B \subseteq J$ with $A \subseteq B$ and a job $j \in J \setminus B$, note that both $\underline{A}_j \subseteq \underline{B}_j$ and $\overline{A}_j \subseteq \overline{B}_j$ must hold. Thus since $a_k > 0$ for all jobs $k \in J$ and p is non-decreasing in S (i.e. $p(S_1) \leq p(S_2)$ for $S_1 \subseteq S_2$), both the left-hand and right-hand terms of the product in $contr(j;B)$ must be at least as large as those of $contr(j;A)$, meaning that $contr(j;A) \leq contr(j;B)$, the desired result. □

Unfortunately, the function C does not appear to fit any classification in terms of modularity. In the simplest case with only a single MP included in the schedule, C becomes

$$C(S) = p(S) + \bar{p}(S) + \ell, \text{ where } \bar{p}(S) := p(J \setminus S).$$

The structure of the function $\bar{p}$ is largely opposite to that of p; $\bar{p}$ is submodular (i.e. $\bar{p}(A+j) - \bar{p}(A) \geq \bar{p}(B+j) - \bar{p}(B)$ for $A \subseteq B$ and $j \in J \setminus B$) and non-increasing in S (i.e. $\bar{p}(S_1) \geq \bar{p}(S_2)$ for $S_1 \subseteq S_2$). Thus $C(S)$, as the sum of a supermodular non-decreasing and a submodular non-increasing function, appears to have no useful structural properties in terms of modularity or monotonicity.

4.3.3 Deciding whether to schedule a single MP is just as hard as solving the problem

In order to rule out the inclusion of a single available MP of length ℓ in a makespan-optimal schedule, we must demonstrate that there does not exist any subset $S \subseteq J$ for which $C(S) \leq C(J)$, i.e. that

$$\min_{\emptyset \neq S \subset J} \{p(S) + \bar{p}(S)\} + \ell \leq p(J).$$

This is clearly equivalent to solving the problem outright.

When we turn our attention to schedules involving more than one MP, meaning C needs two or more sets as input, the situation becomes even murkier. Makespan optimization becomes a game of "give and take"—removing a job j from a subset S_1 will reduce its processing time by $contr(j; S_1)$, but j must then be reassigned to another subset S_2, leading to an increase of $contr(j; S_2)$ in its processing time. While we have yet to discover a polynomial-time means of properly managing these dynamics to minimize makespan, if it exists, the next section proposes several heuristics that greedily produce optimal or near-optimal schedules under certain conditions.

4.4 Integer Programming Formulations

In this section, we present two integer linear programs that may be useful for solving the problem of interest. The first is an exact formulation for the most general version of the problem with up to K MPs, while the second is a "proxy" formulation for the special case with a single available MP and only two distinct deterioration rates a_1 and a_2.

4.4.1 Set partitioning formulation

From equation (4.3) it is clear that minimizing schedule makespan with K available MPs is essentially the problem of finding a makespan-optimal way to partition the job set J into at most $K + 1$ subsets. Thus, using a binary variable z_S to model the decision to include a subset $S \subseteq J$ in the

partition, we can formulate the problem as

$$\min \sum_{S \in 2^J} p(S) z_S + \ell \left(\sum_{S \in 2^J} z_S - 1 \right) \tag{4.4a}$$

$$\text{s.t.} \sum_{S \in 2^J} e_j^S z_S = 1 \qquad \forall j \in J \tag{4.4b}$$

$$\sum_{S \in 2^J} z_S \leq K + 1 \tag{4.4c}$$

$$z_S \in \{0, 1\} \qquad \forall S \in 2^J \tag{4.4d}$$

where 2^J is the power set of J, and e_j^S is 1 if $j \in S$ and 0 otherwise. The first term of the objective function simply sums the weights of the chosen subsets, while the second term ensures that the quantity ℓ is added an appropriate number of times based on the number of subsets selected. Constraints (4.4b) ensure that each job is scheduled exactly once, and constraint (4.4c) limits the number of subsets we can choose based on the number of available MPs.

If we define $w_S = p(S) + \ell$, we see that this looks quite similar to an IP formulation for the Weighted Set Cover problem, with the main difference being an additional constraint on the cardinality of the chosen cover. Note also that constraints (4.4b) would normally have the form "$\geq$" rather than "$=$", but in this particular setting any collection of subsets that covers a job more than once is sub-optimal and need not be considered. For example, if the cover $\{S_1, S_2\}$ has $j \in S_1 \cap S_2$, makespan can be reduced without losing feasibility by replacing S_1 with $S_1 \setminus j$ or doing the same for S_2.

While this may seem convenient, it is only the case because we are forced to account for all 2^n possible subsets of J in this formulation. As run-times for heuristic algorithms for solving Weighted Set Cover generally hinge on the size m of the collection of sets from which we may choose, we are probably better served by seeking other means of obtaining quality solutions.

Column Generation

Column generation is a common iterative approach for solving linear and integer programs with exponentially many variables. At each iteration, we compute the optimal dual values for a version of the LP with only a chosen subset of the variables included, then use these dual values in solving a subproblem over the remaining variables to see if any of them has an attractive reduced cost. If one does, we add it to our subset of variables and repeat the process; otherwise we terminate with an optimal solution to the full LP.

If we replace the integrality requirements (4.4d) with non-negativity constraints, the LP dual of our formulation is given by

$$\max \sum_{j \in J} u_j + (K+1)v - \ell \tag{4.5a}$$

$$\text{s.t.} \sum_{j \in J} e_j^S u_j + v \leq w_S \qquad \forall S \in 2^J \tag{4.5b}$$

$$\mathbf{u} \text{ unrestricted}, \; v \leq 0. \tag{4.5c}$$

Here the number of variables is manageable while the number of constraints is large, but if we only choose to consider a relatively small collection $\mathcal{C}$ of subsets of J, an optimal dual solution $(\bar{\mathbf{u}}, \bar{v})$ is easily obtained. The reduced cost of any variable z_S in terms of $(\bar{\mathbf{u}}, \bar{v})$ is given by

$$w_S - \sum_{j \in J} e_j^S \bar{u}_j - \bar{v},$$

and since we are in a minimization setting, our column generation sub-problem seeks the subset $S \in J \setminus \mathcal{C}$ with the minimum reduced cost. If we assume that jobs are indexed from 1 to n in ascending order of their h_j values, we can use binary variables x_j modeling the inclusion of job j

in our subset to express the subproblem as

$$SP := \min_{\mathbf{x}\in\{0,1\}^n} p_n x_n + \sum_{j=1}^{n-1} p_j x_j \left(\prod_{k=j+1}^{n} (1 + a_k x_k) \right) + \ell - \sum_{j=1}^{n} \bar{u}_j x_j - \bar{v},$$

which is, unfortunately, highly non-linear as it includes products of $\mathcal{O}(n)$ binary variables.

As mentioned above, Wang et al. [41] generate columns using a different IP formulation that results in a linear pricing problem.

4.4.2 An Approximate Formulation when $a_j \in \{a_1, a_2\}$ and $K = 1$

Given that all indications are that the considered problem is NP-Hard, we next investigate the difficulty of solving a relatively simple special case. Specifically, we examine the case involving only two distinct deterioration rates a_1 and a_2 and a single available MP and propose a mixed integer linear program with more manageable numbers of variables and constraints whose solution provides a high-quality, but not necessarily optimal, schedule. Allowing for only two distinct deterioration rates mean that the coefficient for each processing time in the makespan calculation will consist of only two values, $1+a_1$ and $1+a_2$, raised to variable powers. Additionally, including only one available MP makes the decision of whether or not to schedule it relatively simple; we need only solve the problem assuming the MP will be scheduled and compare this to schedule makespan with no MP (i.e. $p(J)$).

With only one MP available for scheduling, we can model feasible schedules using binary variables x_{ij} that take value 1 when jobs i and j are scheduled on the same side of the MP (i.e.

both before or both after) and 0 otherwise. Imposing the constraints

$$x_{ij} - x_{ik} - x_{jk} \geq -1 \tag{4.6a}$$

$$x_{ik} - x_{ij} - x_{jk} \geq -1 \tag{4.6b}$$

$$x_{jk} - x_{ij} - x_{ik} \geq -1 \tag{4.6c}$$

$$x_{ij} + x_{ik} + x_{jk} \geq 1, \tag{4.6d}$$

for each "triple" of jobs (i, j, k) ensures that the schedule has the appropriate structure (with no more than two job groupings). If we let a and b represent $1 + a_1$ and $1 + a_2$, respectively, and assume that the MP will be scheduled, we can express schedule makespan as

$$\sum_{j=1}^{n} p_j a^{m_j} b^{n_j} + \ell, \tag{4.7}$$

where we again assume that jobs are indexed in ascending order of their h_j values, and the exponents m_j and n_j are defined as

$$m_j := \sum_{\substack{k:j \prec k \\ a_k = a_1}} x_{jk} \text{ and } n_j := \sum_{\substack{k:j \prec k \\ a_k = a_2}} x_{jk},$$

with $m_n = n_n = 0$. We can "linearize" the first piece of (4.7) by taking the base-a logarithm of the exponential parts of each term in the sum, which yields the proxy formulation below:

$$\begin{aligned} \min \sum_{j=1}^{n} p_j \left(m_j + \frac{n_j}{\log_b(a)} \right) + \ell \\ \text{s.t. (4.6a)} - \text{(4.6d)} \qquad \forall (i,j,k) : i < j < k \\ x_{ij} \in \{0, 1\} \qquad \forall (i,j) : i < j \end{aligned}$$

This formulation is attractive because of its size—its numbers of variables and constraints are both

polynomial in n ($\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively). An obvious drawback is that we no longer have an exact formulation; jobs may have proportionally more or less influence over the linearized objective value than they had over the original. Our initial computational experience presented in the next section suggests that while an optimal schedule according to this proxy objective function is in many instances not truly optimal, its makespan is generally "good", i.e. close to C^*_{max}.

4.5 Greedy Heuristic Algorithms

In this section, we outline several greedy algorithms that appear, in our computational study, to yield high-quality schedules, along with a simple swap-based local search technique that we can use to improve these schedules. All five algorithms presented are for the general case with K available MPs.

4.5.1 A Randomized Algorithm

We begin by introducing a "baseline" algorithm of sorts—a randomized algorithm that partitions J into up to $K+1$ pieces by assigning jobs to subsets uniformly at random. The pseudocode is as follows:

Algorithm 4 Randomized Assignment Algorithm

1: Initialize: $S_k \leftarrow \emptyset$ for $k = 1, 2, \ldots K+1$
2: **for** $j = 1$ to n **do**
3: Draw value k^* uniformly at random from $\{1, 2, \ldots K+1\}$.
4: $S_{k^*} \leftarrow S_{k^*} \cup \{j\}$
5: **return** $S_1, S_2, \ldots, S_{K+1}$

In the absence of a computationally efficient exact algorithm, we measure the effectiveness of our forthcoming greedy algorithms by comparing their performance against that of this randomized algorithm; if there is not a significant difference, then a greedy algorithm is considered to be of little value and ineffective.

4.5.2 Sorting Algorithms

The next three algorithms we will discuss are sorting algorithms; they all start by sorting the jobs of J into a specific order according to some metric $v(j)$, and then assign each of them in turn to whichever subset leads to the smallest increase in schedule makespan. The general format for these algorithms is as follows:

Algorithm 5 Sorting Algorithm under Metric $v(j)$

Initialize: $S_k \leftarrow \emptyset$, $y_k \leftarrow 0$ for $k = 1, 2, \ldots K+1$
Sort J in ascending (descending) order of $v(j)$
for $j \in J$ **do**
    Determine $k^* = \arg\min_k \{contr(j; S_k) + \ell(1 - y_k)\}$.
    $S_{k^*} \leftarrow S_{k^*} \cup \{j\}$
    if $y_{k^*} = 0$ **then**
        $y_{k^*} = 1$
return $S_1, S_2, \ldots, S_{K+1}$

The three sorting algorithms we have chosen to focus on are

1. sort jobs in ascending order of $v(j) = h_j$,
2. sort jobs in descending order of $v(j) = p_j + a_j$, and
3. sort jobs in descending order of $v(j) = contr(j; J)$.

The first of these was chosen with the idea that placing jobs with smaller h_j values first would give us a good sense for how to properly schedule jobs with a larger h_j values whose deterioration rates have a great deal of influence on makespan. The second was chosen because early computational experience showed that jobs with relatively large p_j and a_j values (but not necessarily a large h_j value) seem to have an outsize influence on schedule makespan and should thus be placed first. The same appeared to be true for jobs with a large contribution to schedule makespan without maintenance.

The computational results compiled below indicate that all three of these algorithms perform well in practice, producing schedules that may differ in terms of job grouping but have similar makespans. However, for each of the first two algorithms there exist relatively simple (though perhaps unrealistic) instances that lead to large optimality gaps. The first sorting method produces a schedule with a makespan more than 100 times as large as the optimal makespan for the simple instance with $p = [10, 15, 2000],\ a = [10, 10, 1000]$ and a single MP with length less than 100. The second sorting method produces a schedule with a makespan more than four times as large as the optimal makespan for the simple instance with $p = [10, 500, 1000],\ a = [1000, 500, 10]$ and a single MP with length less than 100.

4.5.3 A Greedy Improvement Algorithm

For this greedy algorithm, we start from a schedule without a maintenance period, i.e. the simplest partition $S_1 = J$. The idea is to dynamically add MPs to the schedule (and consequently new sets to the partition) when doing so reduces makespan, and to "cascade" jobs from older sets to newer ones until no more improvements can be made. We begin by attempting improving moves from S_1 into S_2, deciding to add a MP if the moves collectively lead to a reduction in makespan (i.e. $C(S_1^{old}) - C(S_1^{new}, S_2) > 0$). Next, we consider introducing S_3, "cascading" improving jobs first from S_1 to S_2 and then from S_2 to S_3 to decide if another MP should be added. This process continues until we reach a step where no improving moves are found. More formally, the algorithm is as follows:

Algorithm 6 Greedy Algorithm

1: Initialize: $S_1 \leftarrow J,\ S_k \leftarrow \emptyset$ for $k = 2, \ldots, K+1,\ K' \leftarrow 1$
2: success $\leftarrow$ FALSE
3: **for** $k = 1, \ldots, K'$ **do**
4: **if** $k < K'$ **then**
5: Choose $j^* \in \arg\max_{j \in S_k} \{C(S_k, S_{k+1}) - C(S_k - j, S_{k+1} + j)\}$
6: $\rho \leftarrow C(S_k, S_{k+1}) - C(S_k - j^*, S_{k+1} + j^*)$
7: **if** $\rho \geq 0$ **then**
8: $S_k \leftarrow S_k - j^*,\ S_{k+1} \leftarrow S_{k+1} + j^*$
9: success $\leftarrow$ TRUE
10: Go to line 5
11: **else if** $k = K'$ **then**
12: new_set $\leftarrow$ FALSE, $T_0 \leftarrow S_{K'},\ T_1 \leftarrow \emptyset$
13: Choose $j^* \in \arg\max_{j \in T_0} \{C(T_0) - C(T_0 - j, T_1 + j)\}$
14: $\rho \leftarrow C(T_0) - C(T_0 - j^*, T_1 + j^*) - \ell$
15: **if** $\rho \geq 0$ **then**
16: $T_0 \leftarrow T_0 - j^*,\ T_1 \leftarrow T_1 + j^*$
17: Go to line 13
18: $\sigma \leftarrow C(T_0) - C(T_0, T_1)$
19: **if** $\sigma > \ell$ **then**
20: $S_{K'} \leftarrow T_0,\ S_{K'+1} \leftarrow T_1$
21: new_set $\leftarrow$ TRUE, success $\leftarrow$ TRUE
22: **if** new_set = TRUE **then**
23: $K' \leftarrow K' + 1$
24: **if** success = TRUE **then**
25: Go to line 2
26: **else**
27: **return** $S_1, S_2, \ldots S_{K'+1}$

Results presented later in the section for test instances involving a single MP indicate that this algorithm generally produces high-quality schedules with makespans within one or two percentage points of the optimal makespan. However, computational investigations have shown that this is will not always be the case; we have (with some difficulty) been able to construct 5-job instances for which the algorithm yields a solution with makespan more than 3 times as large as the optimal

makespan, e.g., $p = [2.22, 8.56, 33.59, 0.44, 0.44],\ a = [200, 622.17, 207, 2.67, 2.66]$, and as the number of jobs increases this factor can balloon to more than 10. These "bad" instances all share some common characteristics:

- job deterioration rates are at least as large as, and generally much larger than, job base processing times,
- the largest h_j ratio is at least a factor of 10 larger than the smallest ratio, and
- one or two jobs have a much larger $p_j + a_j$ value than the other jobs.

It should be noted, however, that the sorting algorithms presented above all return the optimal schedule for the "bad" 5-job instance. When instances are "better behaved", e.g., with processing times always exceeding deterioration rates and no outliers in terms of $p_j + a_j$ value, it seems for the most part that "greed is good".

4.5.4 Local Search

To be able to produce high-quality solutions, we complement the greedy construction heuristics with a swap-based local search algorithm. It searches for pairs of jobs in different MP groups that can be swapped to improve makespan until none can be found (i.e. we have a swap-optimal partition).

Whereas exhaustive search is viable for instances with a single available MP, it may become computationally prohibitive for instances with a large number of available MPs. In such situations, it may be better to randomly select pairs of groups to check for swaps a set number of times.

Algorithm 7 Swap Neighborhood Search Algorithm

1: Start with a partition $(S_1, S_2, \ldots, S_{K+1})$
2: **while** $\exists\, j_1 \in S_{k_1}, j_2 \in S_{k_2}$ s.t. $C(S_{k_1} - j_1 + j_2, S_{k_2} - j_2 + j_1) < C(S_{k_1}, S_{k_2})$ **do**
3: $\quad S_{k_1} \leftarrow S_{k_1} - j_1 + j_2,\ S_{k_2} \leftarrow S_{k_2} - j_2 + j_1$
4: **return** $(S_1, S_2, \ldots, S_{K+1})$

4.5.5 Computational Analysis with $K = 1$

Greedy Algorithms

We implemented the aforementioned randomized, sorting, and greedy algorithms in Python and tested them on 66 randomly generated families of instances with a single available MP of length $\ell = 1$ (so that it will always be worthwhile to schedule it somewhere), each comprised of 100 instances with the same number of jobs and random job processing times and deterioration rates drawn from the same uniform distributions.

We arrived at the 66 families of instances by varying the number of jobs (from 10 up to 20), the distribution for processing times ($U[1, 20]$, $U[1, 40]$, and $U[1, 60]$), and the distribution for deterioration rates ($U[0, 1]$, $U[0, 3]$). We chose these distributions and instance sizes to reflect the kinds of proportional relationships between base processing times and deterioration rates that we think would be applicable yet interesting to explore. Mosheiov [47] and others have noted that as the number of jobs gets large, base processing times have less and less of an effect on makespan until processing times of the form $p_j(t) = a_j t$ provide an adequate approximation. Thus we strove in our sample instances to maximize the impact of base processing times by keeping deterioration rates relatively small and limiting instance size. Another motivation for capping instance sizes at 20 jobs was our desire to compare the makespans of the schedules obtained using our heuristics to the optimal makespan; we computed the optimal value for each instance by enumeration, which became computationally burdensome for large values of n on account of there existing 2^n feasible schedules.

The six tables below display average optimality gaps for each heuristic over the 100 instances of each instance family. Each heuristic introduced was both implemented on its own and combined with swap neighborhood search. We abbreviate our sorting algorithms as S1, S2, and S3 (numbered according to the order in which we first presented them), our greedy improvement algorithm as G, and the swap neighborhood search algorithm as LS. Each table corresponds to a unique pairing

of processing time and deterioration rate distributions, with each row displaying the results for a given instance size. Here we define the optimality gap as

$$\text{Optimality Gap} = \frac{C_{max}(H) - C^*_{max}}{C^*_{max}},$$

where $C_{max}(H)$ is the heuristic solution's objective value and C^*_{max} is the optimal objective value.

Table 4.1: $U[1, 20]$ processing times and $U[0, 1]$ deterioration rates

Jobs	**Random**		**Greedy**		**Sorting 1**		**Sorting 2**		**Sorting 3**	
	R	**R + LS**	**G**	**G + LS**	**S1**	**S1 + LS**	**S2**	**S2 + LS**	**S3**	**S3 + LS**
10	0.20254	0.06099	0.00817	0.00290	0.00254	0.00065	0.00575	0.00045	0.00689	0.00110
11	0.24512	0.04628	0.00994	0.00425	0.00323	0.00082	0.00562	0.00081	0.00677	0.00101
12	0.21455	0.03200	0.01002	0.00236	0.00269	0.00046	0.00560	0.00044	0.00834	0.00102
13	0.22356	0.02936	0.01102	0.00372	0.00239	0.00089	0.00604	0.00111	0.00699	0.00151
14	0.25027	0.03208	0.01206	0.00368	0.00163	0.00037	0.00597	0.00109	0.00845	0.00091
15	0.41074	0.08132	0.01554	0.00504	0.00231	0.00108	0.00735	0.00120	0.00856	0.00185
16	0.33056	0.02403	0.01594	0.00534	0.00216	0.00074	0.00637	0.00094	0.00846	0.00121
17	0.36804	0.02039	0.01546	0.00478	0.00179	0.00084	0.00739	0.00105	0.00793	0.00131
18	0.34739	0.02808	0.01653	0.00451	0.00181	0.00072	0.00744	0.00121	0.00872	0.00133
19	0.27725	0.01207	0.01981	0.00555	0.00175	0.00075	0.00857	0.00131	0.00998	0.00159
20	0.51689	0.02534	0.02185	0.00650	0.00133	0.00061	0.00733	0.00104	0.00867	0.00156

Table 4.2: $U[1, 20]$ processing times and $U[0, 3]$ deterioration rates

Jobs	Random		Greedy		Sorting 1		Sorting 2		Sorting 3	
	R	**R + LS**	**G**	**G + LS**	**S1**	**S1 + LS**	**S2**	**S2 + LS**	**S3**	**S3 + LS**
10	1.57957	0.47051	0.02320	0.01148	0.00701	0.00161	0.01353	0.00193	0.02302	0.00313
11	2.27459	0.51599	0.03121	0.01594	0.00825	0.00164	0.01859	0.00177	0.02580	0.00330
12	1.83591	0.36046	0.03306	0.01274	0.00809	0.00150	0.02229	0.00204	0.02673	0.00245
13	1.92182	0.33143	0.03364	0.01695	0.00737	0.00159	0.02127	0.00192	0.02959	0.00259
14	2.26221	0.22198	0.02976	0.01434	0.00844	0.00173	0.02111	0.00198	0.02637	0.00321
15	2.17007	0.28360	0.03630	0.01628	0.00728	0.00139	0.02026	0.00221	0.02669	0.00280
16	2.70497	0.23210	0.03518	0.01517	0.00704	0.00081	0.01812	0.00255	0.03167	0.00294
17	3.46733	0.48788	0.03672	0.01614	0.00457	0.00080	0.01794	0.00252	0.03448	0.00302
18	5.07705	0.27269	0.03656	0.01401	0.00610	0.00084	0.01905	0.00191	0.03061	0.00272
19	9.33741	0.54197	0.03473	0.01219	0.00454	0.00086	0.02167	0.00218	0.02607	0.00224
20	3.97756	0.18312	0.03557	0.01154	0.00328	0.00059	0.02494	0.00236	0.03518	0.00272

Table 4.3: $U[1, 40]$ processing times and $U[0, 1]$ deterioration rates

Jobs	Random		Greedy		Sorting 1		Sorting 2		Sorting 3	
	R	**R + LS**	**G**	**G + LS**	**S1**	**S1 + LS**	**S2**	**S2 + LS**	**S3**	**S3 + LS**
10	0.20117	0.06718	0.00703	0.00214	0.00205	0.00066	0.00476	0.00062	0.00656	0.00121
11	0.22817	0.05210	0.00953	0.00347	0.00279	0.00089	0.00531	0.00084	0.00648	0.00099
12	0.23055	0.04630	0.01055	0.00390	0.00310	0.00072	0.00580	0.00059	0.00659	0.00089
13	0.29171	0.03751	0.01179	0.00466	0.00247	0.00067	0.00714	0.00105	0.00872	0.00156
14	0.30107	0.03370	0.01261	0.00399	0.00203	0.00064	0.00554	0.00096	0.00784	0.00124
15	0.37463	0.04246	0.01292	0.00402	0.00218	0.00059	0.00724	0.00129	0.00784	0.00128
16	0.33155	0.05532	0.01393	0.00499	0.00205	0.00059	0.00618	0.00113	0.00913	0.00125
17	0.38454	0.04779	0.01633	0.00562	0.00187	0.00083	0.00665	0.00111	0.00851	0.00149
18	0.42561	0.09097	0.01626	0.00520	0.00155	0.00068	0.00801	0.00111	0.00868	0.00154
19	0.45675	0.03878	0.01850	0.00568	0.00177	0.00067	0.00850	0.00110	0.00885	0.00177
20	0.41362	0.01587	0.01862	0.00602	0.00146	0.00072	0.00651	0.00102	0.00863	0.00154

Table 4.4: $U[1, 40]$ processing times and $U[0, 3]$ deterioration rates

Jobs	Random		Greedy		Sorting 1		Sorting 2		Sorting 3	
	R	**R + LS**	**G**	**G + LS**	**S1**	**S1 + LS**	**S2**	**S2 + LS**	**S3**	**S3 + LS**
10	1.54386	0.66927	0.02584	0.01072	0.00639	0.00143	0.01630	0.00113	0.02184	0.00451
11	1.73637	0.60917	0.02823	0.01104	0.00754	0.00187	0.01968	0.00291	0.02417	0.00293
12	1.79819	0.58195	0.02802	0.01387	0.01169	0.00120	0.01889	0.00194	0.02578	0.00321
13	2.28629	0.33613	0.03370	0.01525	0.00775	0.00186	0.01845	0.00214	0.02774	0.00246
14	2.45204	0.35818	0.03747	0.01489	0.00560	0.00133	0.01943	0.00267	0.02807	0.00358
15	3.35935	0.48650	0.03798	0.01528	0.00712	0.00132	0.02234	0.00295	0.02161	0.00279
16	4.05388	0.93549	0.03359	0.01537	0.00703	0.00090	0.02257	0.00262	0.02849	0.00252
17	4.55802	0.98348	0.03730	0.01556	0.00531	0.00119	0.02014	0.00239	0.02963	0.00365
18	6.84955	2.27050	0.03904	0.01672	0.00478	0.00099	0.02059	0.00232	0.02822	0.00247
19	4.38865	0.51547	0.03440	0.01392	0.00557	0.00070	0.02481	0.00167	0.02960	0.00256
20	3.74134	0.15502	0.03320	0.01508	0.00372	0.00062	0.02370	0.00259	0.02736	0.00272

Table 4.5: $U[1, 60]$ processing times and $U[0, 1]$ deterioration rates

Jobs	Random		Greedy		Sorting 1		Sorting 2		Sorting 3	
	R	**R + LS**	**G**	**G + LS**	**S1**	**S1 + LS**	**S2**	**S2 + LS**	**S3**	**S3 + LS**
10	0.19995	0.06619	0.00695	0.00213	0.00192	0.00061	0.00483	0.00054	0.00630	0.00119
11	0.21757	0.05078	0.00944	0.00331	0.00247	0.00081	0.00523	0.00091	0.00650	0.00106
12	0.23306	0.04568	0.01056	0.00379	0.00286	0.00084	0.00536	0.00059	0.00669	0.00092
13	0.27929	0.03605	0.01099	0.00412	0.00261	0.00073	0.00671	0.00106	0.00859	0.00162
14	0.28978	0.03288	0.01252	0.00416	0.00202	0.00069	0.00555	0.00094	0.00781	0.00125
15	0.36734	0.04170	0.01310	0.00417	0.00206	0.00079	0.00701	0.00126	0.00764	0.00139
16	0.33972	0.05366	0.01405	0.00495	0.00221	0.00057	0.00567	0.00091	0.00966	0.00147
17	0.37218	0.04638	0.01645	0.00561	0.00182	0.00079	0.00641	0.00110	0.00817	0.00144
18	0.45690	0.08685	0.01579	0.00527	0.00152	0.00070	0.00815	0.00133	0.00899	0.00159
19	0.45371	0.03711	0.01888	0.00601	0.00196	0.00074	0.00791	0.00103	0.00933	0.00185
20	0.43218	0.01518	0.01866	0.00582	0.00132	0.00070	0.00622	0.00105	0.00886	0.00153

Table 4.6: $U[1, 60]$ processing times and $U[0, 3]$ deterioration rates

Jobs	Random		Greedy		Sorting 1		Sorting 2		Sorting 3	
	R	R + LS	G	G + LS	S1	S1 + LS	S2	S2 + LS	S3	S3 + LS
10	1.52948	0.65798	0.02457	0.00897	0.00597	0.00091	0.01566	0.00107	0.02119	0.00420
11	1.66945	0.59044	0.02828	0.01148	0.00704	0.00177	0.01971	0.00233	0.02153	0.00322
12	1.70853	0.57370	0.02726	0.01263	0.00994	0.00121	0.01970	0.00208	0.02426	0.00362
13	2.23055	0.32366	0.03312	0.01406	0.00750	0.00137	0.01958	0.00261	0.02843	0.00347
14	2.82487	0.35132	0.03735	0.01310	0.00533	0.00125	0.01830	0.00267	0.02907	0.00340
15	3.52513	0.47094	0.03860	0.01693	0.00626	0.00137	0.02102	0.00286	0.02512	0.00310
16	3.85065	0.91189	0.03363	0.01593	0.00815	0.00099	0.02349	0.00265	0.02651	0.00268
17	6.90145	0.94497	0.03804	0.01527	0.00515	0.00136	0.01920	0.00227	0.02745	0.00251
18	7.74517	2.15159	0.03868	0.01700	0.00499	0.00092	0.02147	0.00218	0.02537	0.00275
19	4.51699	0.49841	0.03514	0.01412	0.00558	0.00072	0.02435	0.00182	0.03240	0.00287
20	3.76280	0.14731	0.03403	0.01458	0.00379	0.00054	0.02114	0.00289	0.02409	0.00276

While these results suggest that sorting and placing jobs according to their h_j values is the clear winner in terms of average quality, there are certain instances for which one of the other sorting algorithms or even the greedy algorithm is best. It also appears that local search has more of a positive impact on the optimality gaps for the sorting algorithms than it does for the greedy algorithm. It is also worth noting (though not very surprising) that the random algorithm is much more negatively affected by a larger range for deterioration rates than any of the greedy algorithms.

Greedy Algorithms and Approximate IP Formulation

To assess the effectiveness of our approximate IP formulation introduced in the previous section relative to our greedy heuristics, we generated and tested 10 additional families (ranging in size from 11 jobs to 20) of 100 instances each with processing times drawn from $U[1, 60]$ and only two distinct deterioration rates drawn from $U[0, 1]$. The formulation was implemented in Python and solved with Gurobi; the table below displays average optimality gaps computed in the same way

as in the tables above:

Table 4.7: $U[1, 60]$ processing times and two distinct deterioration rates drawn from $U[0, 1]$

Jobs	**Random**	**Greedy**	**Sorting 1**	**Sorting 2**	**Sorting 3**	**Approx IP**
11	0.24728	0.00517	0.00271	0.00349	0.00380	0.00291
12	0.22378	0.00842	0.00263	0.00258	0.00370	0.00296
13	0.40439	0.00861	0.00433	0.00422	0.00463	0.00326
14	0.22396	0.01031	0.00267	0.00426	0.00387	0.00304
15	0.35696	0.00930	0.00328	0.00526	0.00495	0.00226
16	0.49215	0.00864	0.00354	0.00382	0.00421	0.00268
17	0.23081	0.01073	0.00380	0.00547	0.00412	0.00220
18	0.41583	0.01461	0.00285	0.00408	0.00469	0.00203
19	0.47724	0.01511	0.00344	0.00640	0.00495	0.00174
20	0.75742	0.01281	0.00326	0.00478	0.00451	0.00147

For instances with two distinct deterioration rates, the approximate IP formulation appears to suggest an excellent job partitioning, better than the greedy heuristics, especially the number of jobs increases. This may be due in part to the aforementioned diminishing effect of job processing times (which remain unaltered in our objective function) as instance size increases. However, larger instances also mean increased amounts of computation time and power needed to solve the approximate IP, which limits the usefulness of this approach.

Larger Numbers of Jobs

As instance sizes grow past 20 jobs, it becomes increasingly difficult both to compute the optimal makespan by enumeration and to solve our approximate IP formulation to optimality. We can still, however, get some sense of the quality of the schedules produced by our greedy algorithms by comparing them to the schedules generated by the Randomized Assignment algorithm. The table

below gives the average value of $\frac{C_{max}(H)}{C_{max}(R)}$ for each heuristic H over 100 instances with processing times drawn from $U[1, 60]$ and deterioration rates from $[0, 1]$.

Table 4.8: Average heuristic makespans as a fraction of Randomized Assignment makespan when processing times and deterioration rates drawn from $U[1, 60]$, $U[0, 3]$ respectively

Jobs	**Greedy**	**Sorting 1**	**Sorting 2**	**Sorting 3**
10	0.679	0.666	0.673	0.677
20	0.487	0.474	0.482	0.484
50	0.345	0.336	0.341	0.346
75	0.287	0.279	0.283	0.287
100	0.217	0.210	0.213	0.218
125	0.240	0.233	0.237	0.239
150	0.296	0.288	0.291	0.300

Interestingly, the heuristics appear most effective relative to random schedules for instances of size 100, with smaller average improvements for both the 75- and 125-job instances. This trend is likely a result of interplay between the two observations below:

- as the number of jobs increases, so does the potential for "unlucky" random assignments to have makespans many times larger than the optimal makespan, but
- as the number of jobs increases, it becomes increasingly likely that the highest-quality schedules will have equal numbers of jobs on either side of the MP, which is the expected behavior of the Randomized Assignment algorithm.

4.6 Conclusions

In this chapter, we have examined single machine scheduling problems with linear start-time dependent processing times from several different angles. While no single one of our results or

insights is truly ground-breaking, our hope is that the assemblage of work presented in this chapter will be helpful and instructive for future scholarship and practical applications involving problems of this type.

While we have yet to discover a polynomial-time reduction of a known NP-Hard problem to this one, we seriously doubt that a polynomial-time algorithm for solving it exists, especially in light of the result discussed earlier for processing times of the seemingly simpler form $p_j(t) = a_j t$. However, the performance of our five proposed heuristics on instances with up to 20 jobs and the way our greedy solutions compare to randomly obtained solutions for larger instances suggest that there are simple, intuitive ways to obtain high-quality solutions for instances that we believe could be used to reflect realistic scheduling scenarios, e.g. astronaut fatigue or factory downtime for repairs.

REFERENCES

[1] M. Gendreau, A. Hertz, G. Laporte, and M. Stan, "A generalized insertion heuristic for the traveling salesman problem with time windows," *Operations Research*, vol. 46, no. 3, pp. 330–335, 1998.

[2] J. Desrosiers, Y. Dumas, M. M. Solomon, and F. Soumis, "Chapter 2 time constrained routing and scheduling," in *Network Routing*, ser. Handbooks in Operations Research and Management Science, M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, Eds., vol. 8, Elsevier, 1995, pp. 35 –139.

[3] N. Ascheuer, "Hamiltonian path problems in the on–line optimization of flexible manufacturing systems," *PhD Thesis, Technical University of Berlin*, 1995.

[4] O. Bräysy, P. Nakari, W. Dullaert, and P. Neittaanmäki, "An optimization approach for communal home meal delivery service: A case study," *Journal of Computational and Applied Mathematics*, vol. 232, no. 1, pp. 46 –53, 2009.

[5] M. Savelsbergh, "Local search in routing problems with time windows," *Annals of Operations Research*, vol. 4, no. 1, pp. 285–305, 1985.

[6] N. Christofides, A. Mingozzi, and P. Toth, "State-space relaxation procedures for the computation of bounds to routing problems," *Networks*, vol. 11, no. 2, pp. 145–164, 1981.

[7] E. Baker, "An exact algorithm for the time-constrained traveling salesman problem," *Operations Research*, vol. 31, no. 5, pp. 938–945, 1983.

[8] Y. Dumas, E. Desrosiers, M. Gelinas, and M. Solomon, "An optimal algorithm for the traveling salesman problem with time windows," *Operations Research*, vol. 43, no. 2, pp. 367–371, 1995.

[9] A. Mingozzi, L. Bianco, and S. Ricciardelli, "Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints," *Operations Research*, vol. 45, no. 3, pp. 365–377, 1997.

[10] E. Balas and N. Simonetti, "Linear time dynamic-programming algorithms for new classes of restricted tsps: A computational study," *INFORMS Journal on Computing*, vol. 13, no. 1, pp. 56–75, 2001.

[11] C. E. Miller, A. W. Tucker, and R. A. Zemlin, "Integer programming formulation of traveling salesman problems," *J. ACM*, vol. 7, no. 4, pp. 326–329, Oct. 1960.

[12] N. Ascheuer, M. Fischetti, and M. Grötschel, "Solving the asymmetric travelling salesman problem with time windows by branch-and-cut," *Mathematical Programming*, vol. 90, no. 3, pp. 475–506, 2001.

[13] G. Pesant, M. Gendreau, J.-Y. Potvin, and J.-M. Rousseau, "An exact constraint logic programming algorithm for the traveling salesman problem with time windows," *Transportation Science*, vol. 32, no. 1, pp. 12–29, 1998.

[14] F. Focacci, A. Lodi, and M. Milano, "A hybrid exact algorithm for the tsptw," *Informs Journal on Computing*, vol. 14, no. 4, pp. 403–417, 2002.

[15] J. Albiach, J. M. Sanchis, and D. Soler, "An asymmetric {tsp} with time windows and with time-dependent travel times and costs: An exact solution through a graph transformation," *European Journal of Operational Research*, vol. 189, no. 3, pp. 789 –802, 2008.

[16] S. Dash, O. Gunluk, A. Lodi, and A. Tramontani, "A Time Bucket Formulation for the Traveling Salesman Problem with Time Windows," *INFORMS Journal on Computing*, vol. 24, pp. 132–147, 2012.

[17] R. Baldacci, A. Mingozzi, and R. Roberti, "New state-space relaxations for solving the traveling salesman problem with time windows," *INFORMS J. on Computing*, vol. 24, no. 3, pp. 356–371, Jul. 2012.

[18] L. A. Wolsey, *Integer Programming*. Wiley-Interscience, 1998.

[19] G. Desaulniers and S. Irnich, "Shortest path problems with resource constraints," in *Column Generation*, G. Desaulniers, J. Desrosiers, and M. M. Solomon, Eds., Springer US, 2005, ch. 2, pp. 33–65.

[20] A. Langevin, M. Desrochers, J. Desrosiers, and F. Soumis, "A two-commodity flow formulation for the traveling salesman and makespan problem with time windows," *Networks*, vol. 23, no. 7, pp. 631–640, 1993.

[21] M. M. Veloso, A. M. Mulvehill, and M. T. Cox, "Rationale-supported mixed-initiative case-based planning," in *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, ser. AAAI'97/IAAI'97, Providence, Rhode Island: AAAI Press, 1997, pp. 1072–1077, ISBN: 0-262-51095-2.

[22] M. Ai-Chang, J. Bresina, L. Charest, A. Chase, J. C. J. Hsu, A. Jonsson, B. Kanefsky, P. Morris, K. Rajan, J. Yglesias, B. G. Chafin, W. C. Dias, and P. F. Maldague, "Mapgen: Mixed-initiative planning and scheduling for the mars exploration rover mission," *IEEE Intelligent Systems*, vol. 19, no. 1, pp. 8–12, 2004.

[23] F. Nothdurft, G. Behnke, P. Bercher, S. Biundo, and W. Minker, "The interplay of user-centered dialog systems and ai planning," *Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, 2015.

[24] S. Biundo, P. Bercher, T. Geier, F. Müller, and B. Schattenberg, "Advanced user assistance based on ai planning," *Cognitive Systems Research*, vol. 12, no. 3, pp. 219 –236, 2011, Special Issue on Complex Cognition.

[25] K. Myers, S. Smith, D. W. Hildum, P. Jarvis, and R. de Lacaze, "Integrating planning and scheduling through adaptation of resource intensity estimates," in *Proceedings 5th European Conference on Planning*, 2001.

[26] Z. Wang, H.-W. Wang, C. Qi, and J. Wang, "A resource enhanced htn planning approach for emergency decision-making," *Applied Intelligence*, vol. 38, no. 2, pp. 226–238, Mar. 2013.

[27] P. Maldague, A. Ko, D. Page, and T. Starbird, "Apgen: A multi-mission semi-automated planning tool," in *First International NASA Workshop on Planning and Scheduling*, 1998.

[28] S. a. Chien, M. Johnston, J. Frank, M. Giuliano, A. Kavelaars, C. Lenzen, and N. Policella, "A Generalized Timeline Representation, Services, and Interface for Automating Space Mission Operations," in *Proceedings of Space Operations*, American Institute of Aeronautics and Astronautics, 2012, pp. 1–17.

[29] T. Hall, T. LeBlanc, B. Ulman, A. McDonald, P. Gramm, L.-M. Chang, S. Keerthi, D. Kivlovitz, and J. Hadlock, "Onboard short term plan viewer," 2011.

[30] J. J. Marquez, G. Pyrzak, S. Hashemi, K. McMillin, and J. Medwid, "Supporting real-time operations and execution through timeline and scheduling aids," in *43rd International Conference on Environmental Systems*, 2013, p. 3519.

[31] J. J. Marquez, S. Hillenius, B. Kanefsky, J. Zheng, I. Deliz, and M. Reagan, "Increasing crew autonomy for long duration exploration missions: Self-scheduling," in *2017 IEEE Aerospace Conference*, 2017, pp. 1–10.

[32] W. van Wezel and R. Jorna, "Cognition, tasks and planning: Supporting the planning of shunting operations at the netherlands railways," *Cognition, Technology & Work*, vol. 11, no. 2, pp. 165–176, 2009.

[33] D. E. Smith, J. Frank, and A. K. Jonsson, "Bridging the Gap Between Planning and Scheduling," *The Knowledge Engineering Review*, vol. 15, no. 1, pp. 47–83, 2000.

[34] M. Bualat, J. Barlow, T. Fong, C. Provencher, and T. Smith, "Astrobee: Developing a free-flying robot for the international space station," in *AIAA SPACE 2015 Conference and Exposition*. eprint: `https://arc.aiaa.org/doi/pdf/10.2514/6.2015-4643`.

[35] A. R. Pritchett, S. Y. Kim, and K. M. Feigh, "Measuring human-automation function allocation," *Journal of Cognitive Engineering and Decision Making*, vol. 8, no. 1, pp. 52–77, 2014.

[36] M. IJtsma, L. M. Ma, A. R. Pritchett, and K. M. Feigh, "Computational Methodology for the Allocation of Work and Interaction in Human-Robot Teams," *Journal of Cognitive Engineering and Decision Making*, p. 155 534 341 986 948, 2019.

[37] D. Woods, "Cognitive technologies: The design of joint human-machine cognitive systems," *AI magazine*, vol. 6, no. 4, pp. 86–92, 1985.

[38] A. R. Pritchett, R. P. Bhattacharyya, and M. IJtsma, "Computational assessment of authority and responsibility in air traffic concepts of operation," *Journal of Air Transportation*, vol. 24, no. 3, pp. 93–102, 2016.

[39] J. Ullman, "Np-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384 –393, 1975.

[40] Qiang Ji, P. Lan, and C. Looney, "A probabilistic framework for modeling and real-time monitoring human fatigue," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 36, no. 5, pp. 862–875, 2006.

[41] T. Wang, R. Baldacci, A. Lim, and Q. Hu, "A branch-and-price algorithm for scheduling of deteriorating jobs and flexible periodic maintenance on a single machine," *European Journal of Operational Research*, vol. 271, no. 3, pp. 826 –838, 2018.

[42] J. N. Gupta and S. K. Gupta, "Single facility scheduling with nonlinear processing times," *Computers & Industrial Engineering*, vol. 14, no. 4, pp. 387 –393, 1988.

[43] T. Cheng, Q Ding, and B. Lin, "A concise survey of scheduling with time-dependent processing times," *European Journal of Operational Research*, vol. 152, no. 1, pp. 1 –13, 2004.

[44] E. J. Lodree and C. D. Geiger, "A note on the optimal sequence position for a rate-modifying activity under simple linear deterioration," *European Journal of Operational Research*, vol. 201, no. 2, pp. 644 –648, 2010.

[45] M. Ji, C.-J. Hsu, and D.-L. Yang, "Single-machine scheduling with deteriorating jobs and aging effects under an optional maintenance activity consideration," *Journal of Combinatorial Optimization*, vol. 26, Oct. 2013.

[46] K. Rustogi and V. A. Strusevich, "Single machine scheduling with time-dependent linear deterioration and rate-modifying maintenance," *JORS*, vol. 66, pp. 500–515, 2015.

[47] G. Mosheiov, "V-shaped policies for scheduling deteriorating jobs," *Operations Research*, vol. 39, no. 6, pp. 979–991, 1991.

VITA

Will Lassiter was born and raised in Anderson, South Carolina. In 2008 he moved to Gainesville, Florida to study at the University of Florida, where in 2012 he was awarded a bachelor's degree in mathematics. He spent the next two years at Clemson University in Clemson, South Carolina, where in 2014 he received a master's degree in mathematical sciences. Shortly thereafter he moved to Madison, Wisconsin, where he spent a year working for the Epic Systems Corp., a healthcare software company, before deciding in 2015 to continue his education at Georgia Tech, where he has spent the last five years as a doctoral student in operations research within the department of Industrial and Systems Engineering. In 2019, he married the lovely Amy Grady in Seneca, South Carolina and followed her back to Gainesville, Florida, where they currently share a home with a spoiled cat named Gracie. When he isn't thinking about OR problems, Will enjoys tennis, hiking, gaming, crossword puzzles, trivia, and following college sports as an ardent supporter of both the Florida Gators and North Carolina Tar Heels.